

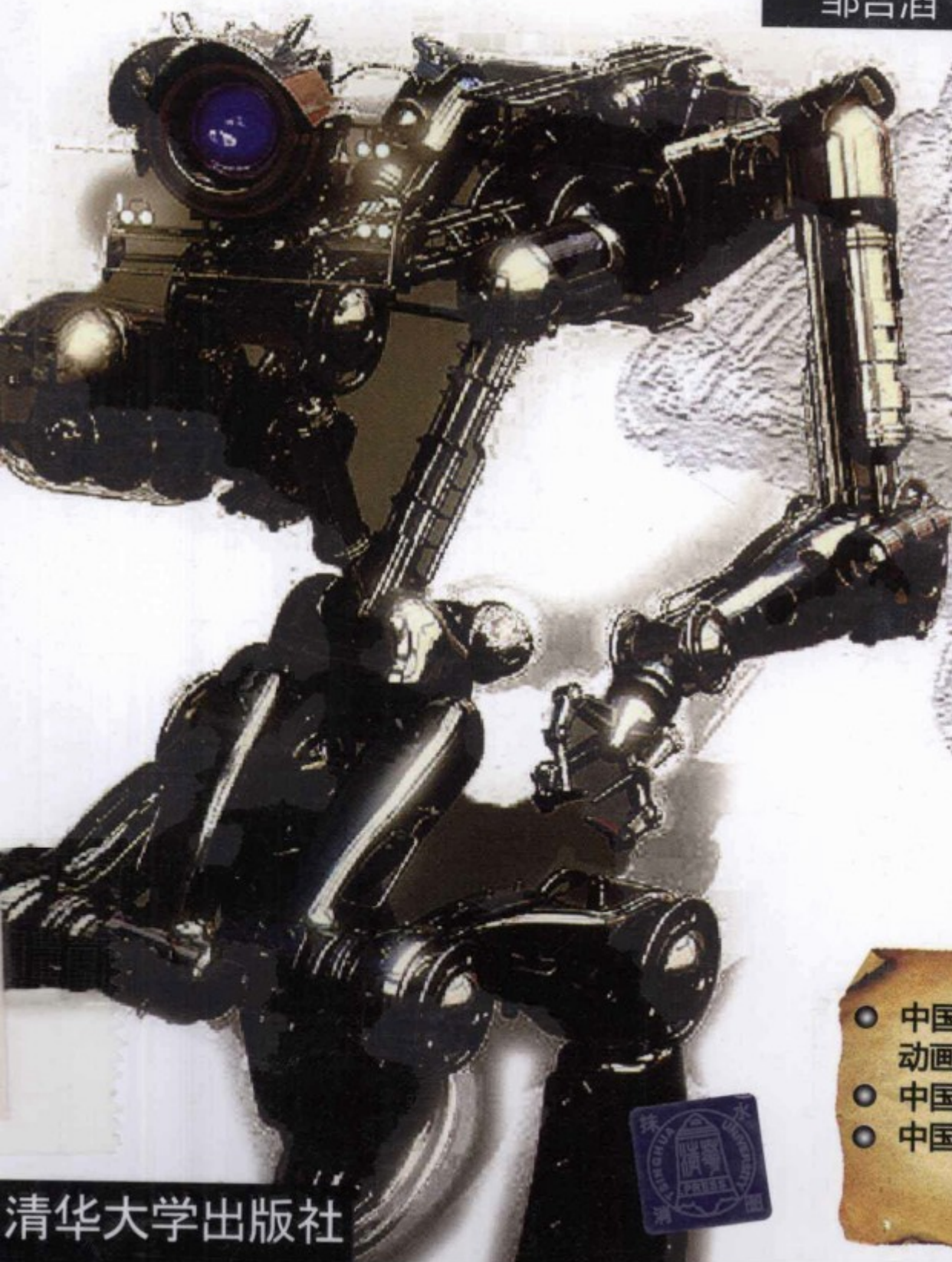
R G D C 游 戏 开 发 课 程 体 系



第九艺术学院——游戏开发系列
COLLEGE OF THE NINTH ART

C++ 游戏编程

邹吉滔 姚 雷 易巧玲 编著



- 中国电影电视技术学会数字特效与三维动画专业委员会
- 中国系统仿真学会数字娱乐专业委员会
- 中国文化创意产业技术创新联盟

推 荐 教 材

清华大学出版社

C++ 游戏编程

第九艺术学院丛书作为RGDC游戏开发课程体系配套教材，由递归教育组织力量，集国内外游戏业内精英人才打造而成。全套丛书共包括：**游戏开发基础**、**游戏美术设计**、**游戏程序开发**3个部分。

关于本书：

本书介绍如何用C++语言进行游戏程序开发。笔者斗胆试着把工作中的一些经验心得以及从无数前辈那里学到的知识整理出来，编写一本C++的基础教学用书。

- ★ C++语言的基础语法部分
- ★ 面向对象编程技术
- ★ 标准模板库的应用

本书级别：专业基础
本书类别：游戏开发



热线:010-82089653
网址:Http://www.recursion.com.cn
MAIL:salcs@recursion.com.cn

ISBN 978-7-302-24591-9



9 787302 245919 >

定价:50.00元

第九艺术学院——游戏开发系列

C++游戏编程

邹吉滔 姚 雷 易巧玲 编著

清华大学出版社

北 京



内 容 简 介

本书介绍如何用 C++ 语言进行游戏程序开发。全书可分为 C++ 语言的基础语法、面向对象编程技术、标准模板库的应用三个部分，共 18 章，主要内容包括：概观程序设计，开发环境简介，基本数据类型，运算符与表达式，程序的结构，宏和编译预处理，数组，函数与程序结构，指针和引用，结构、联合、枚举，类与对象，静态成员与友元，继承与多态，运算符重载，模板，标准模板库，I/O 流，异常处理等。

本书适合游戏开发人员及游戏相关专业师生学习使用，也可供 C++ 编程爱好者参考。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

C++ 游戏编程/邹吉滔，姚雷，易巧玲编著. —北京：清华大学出版社，2011.1

(第九艺术学院——游戏开发系列)

ISBN 978-7-302-24591-9

I. ①C… II. ①邹… ②姚… ③易… III. ①C 语言—程序设计—高等学校—教材 IV. ①TP312

中国版本图书馆 CIP 数据核字(2010)第 254498 号

责任编辑：张彦青 桑任松

封面设计：杨玉兰

责任校对：周剑云

责任印制：杨 艳

出版发行：清华大学出版社

地 址：北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者：北京市清华园胶印厂

装 订 者：三河市金元印装有限公司

经 销：全国新华书店

开 本：185×230 印 张：30.5 字 数：659 千字

版 次：2011 年 1 月第 1 版 印 次：2011 年 1 月第 1 次印刷

印 数：1~4000

定 价：50.00 元

产品编号：023914-01

丛书编委会

主编单位: 北京递归开元教育科技有限公司

主 编: 陈 洪

副主编: 许 影 任 科

编 委: (排名不分先后)

北京理工大学软件学院

丁刚毅

北京工业大学软件学院

蔡建平

西南交通大学软件学院

景 红

上海第九城市信息技术有限公司

黄雪斌

巨人网络集团有限公司

邓 昆

目标软件(北京)有限公司

毛海滨

腾讯公司

姚晓光

文睿信息研究中心

王 涛

创游传媒

孙 辉

中国文化创意产业技术创新联盟

高东旭

北京递归教育科技有限公司

黄 昆

前言

在所有的编程语言中，C++可以说是最为复杂的，它既是一门传统的编程语言，也是一门新的编程语言。

说它是一门传统语言，是因为 C++ 已有 20 多年的历史了；特别是最近 10 年来 C++ 得到了快速的发展。C++ 是计算机软件领域中覆盖面最为广阔的编程语言，并且，与 C++ 相关的智力投入也是其他任何一门语言所无法比拟的。人们对于 C++ 的研究已经远远超出了对于一门编程语言所应有的关注。所以，现在的 C++ 已经非常成熟，有大量的资源(文档、书籍、源代码等)可供我们使用。

说它是一门新的编程语言，是因为在 1998 年 C++ 由国际标准化组织 ISO(International Standards Organization)完成了标准化，从此 C++ 领域有了统一的标准。所有的编译器都将向标准靠拢(或者说，与标准兼容)，这有利于我们写出可移植的 C++ 代码来。同时 C++ 标准也统一了 C++ 标准库，为 C++ 用户提供了最为基本的基础设施。C++ 经历了多年的发展，终于有了一个相对稳定的版本，所以，我们应该用一种新的眼光来看待 C++，而不再简单地把 C++ 认为是 C 语言的超集。通过本书，读者可以重新审视 C++ 语言。

随着编程技术在中国的普及，C++ 语言已经为众多程序设计人员所掌握，相关的书籍也层出不穷，但是现在国内 IT 企业拥有学士、硕士、博士文凭的软件开发人员比比皆是，可由于他们在接受大学教育时就没有对软件开发工作中有关的质量属性(如正确性、健壮性、可靠性、效率、易用性、可读性(可理解性)、可扩展性、可复用性、兼容性、可移植性等)进行深入的分析 and 探讨，并且在实践中运用相关的技术。“高质量”可不是通过读书就能学会的，那需要长期的实践工作以及对工作的成败进行不断的总结，才能实现！

因此有充分的理由疑虑：

(1) 编程老手可能会长期用隐含错误的方式编程(习惯成自然)，发现毛病后都不愿相信那是真的！

(2) 编程高手可以在某一领域写出极有水平的代码，但未必能从全局把握软件质量的方方面面。

事实证明确实如此。大多数“老”程序员的编程技能，质量合格率大约是 10%。很少有人能够写出完全符合质量要求的代码，很多程序员对指针、内存管理一知半解。

在这样的前提之下，笔者斗胆试着把工作中的一些经验心得以及从无数个前辈那里学到的知识整理出来，编写一本 C++ 的基础教学用书，不指望能够把 C++ 的知识全部说出来，

只希望让初学者能够很快看到 C++ 知识基础面貌以及编程的基本方法和风格。本书的一个重要特点是在讲解一些重要知识点的时候，用同一个例子进行说明，通过不断的升级改造同一例程，来说明不同知识点的运用技巧，这些例子主要是在游戏程序设计中用到的一些基础程序，因此对有志作为游戏程序设计者的帮助尤其显著。

本书主要分为 3 个部分，第一部分为 C++ 语言的基础语法部分，主要用来讲述语法知识以及程序的基本结构，运算符、数组、函数、预处理等知识。这部分知识中，有关内存处理部分，需要读者认真体会，第二部分是本书的真正重点，主要讲述面向对象编程技术，这部分的知识，主要通过几个贯穿这部分始末的例子来描述，其中一个重要的例子是一个简单的栈设计及应用，讲解不同知识点的时候，将利用相关的知识点对栈进行升级改造，读者可以充分体会到每个知识点在什么情况下运用可以发挥更大的作用。另外一部分重要的例子是如何在游戏程序设计中利用面向对象编程技术，可以通过设计游戏中的精灵类，来理解面向对象技术的强大作用。本书的第三部分，不作为重点进行讲述，但仍然需要读者认真学习，第三部分主要讲述标准模板库的应用。

本书由邹吉滔，姚雷，易巧玲编著。由于各方面的原因，书中疏漏之处在所难免，恳请广大读者批评指正。

编 者

数字资源
PDG

目 录

第 1 章 概观程序设计	1
1.1 程序设计发展历程	1
1.1.1 什么是计算机程序	1
1.1.2 计算机程序语言的发展历史 ...	2
1.2 程序设计思想	4
1.2.1 结构化程序设计思想	4
1.2.2 面向对象程序设计思想	5
本章小结	8
第 2 章 开发环境简介	9
2.1 Visual Studio .NET 集成开发环境	9
2.1.1 创建项目	10
2.1.2 创建文件	12
2.1.3 项目属性设置	12
2.1.4 编译和运行	13
2.1.5 调试	14
2.1.6 辅助工具	14
2.1.7 解决方案资源管理器	18
2.1.8 类视图	19
2.1.9 文件视图	20
2.1.10 资源视图	20
2.1.11 帮助文档的使用	28
2.2 Linux 下的开发环境	28
2.2.1 Vi 编辑器的基本使用	29
2.2.2 Vi 编辑器的命令	29
2.2.3 Vi 编辑器环境设置	32
2.2.4 g++ 编译程序的方法	33
2.2.5 g++ 编译程序的选项	33
2.2.6 运行应用程序	38
2.2.7 帮助文档的使用	38
2.3 CodeBlocks 集成开发工具介绍	38
2.3.1 创建工程	39
2.3.2 创建文件	39
2.3.3 项目属性设置	39
2.3.4 编译及运行	40
2.4 绘图函数库的使用	41
本章小结	41
第 3 章 基本数据类型	42
3.1 基本程序组成结构	42
3.1.1 一个基本的 C++ 程序	42
3.1.2 基本输入输出	43
3.2 字符集和关键字	47
3.3 C++ 的数据类型概述	49
3.4 基本数据类型	51
3.4.1 整型数据	51
3.4.2 浮点型数据	52
3.4.3 字符型数据	53
3.4.4 bool 类型	56
3.4.5 void 类型	58
3.4.6 常量与变量	58
3.5 类型转换	61
3.5.1 隐式类型转换	62
3.5.2 强制类型转换	64
本章小结	64
第 4 章 运算符与表达式	66
4.1 概述	66
4.2 运算符和表达式	66
4.2.1 运算符和表达式的种类	66
4.2.2 左值和右值	67

4.3 算术运算符和算术表达式.....	68	6.2 头文件包含	108
4.4 自增和自减运算符	69	6.3 条件编译	110
4.5 赋值运算符和赋值表达式.....	71	6.4 其他预处理指令.....	115
4.5.1 赋值运算符与赋值运算	71	本章小结	119
4.5.2 复合赋值运算符.....	72	习题	119
4.5.3 赋值表达式.....	72	第7章 数组	120
4.6 关系运算符和关系表达式.....	73	7.1 为何需要数组.....	120
4.7 逻辑运算符和逻辑表达式.....	75	7.2 声明数组	121
4.7.1 逻辑运算符	75	7.3 访问数组元素.....	122
4.7.2 逻辑表达式.....	76	7.4 数组的初始化.....	124
4.8 sizeof 运算符.....	76	7.5 数组应用举例.....	125
4.9 条件运算符和条件表达式.....	77	7.5.1 选择排序.....	125
4.10 逗号运算符和逗号表达式.....	78	7.5.2 冒泡排序.....	127
4.11 优先性和结合性	79	7.5.3 更多排序算法	129
本章小结	80	7.6 字符串与字符数组.....	131
习题	80	7.7 数组作为函数参数.....	133
第5章 程序的结构	81	7.8 二维数组	134
5.1 顺序结构	81	7.8.1 二维数组的定义	134
5.2 分支结构程序设计	82	7.8.2 二维数组中元素的引用	135
5.2.1 if...else...结构	82	7.8.3 二维数组的初始化	135
5.2.2 switch 语句	86	7.8.4 二维数组程序举例	136
5.2.3 goto 语句.....	89	7.9 多维数组	138
5.3 循环结构程序设计	90	7.9.1 多维数组的定义	139
5.3.1 for 语句	90	7.9.2 多维数组的引用	139
5.3.2 while 语句.....	94	本章小结	141
5.3.3 do-while 语句.....	96	习题	141
5.3.4 循环的嵌套.....	97	第8章 函数与程序结构.....	142
5.4 break、continue 语句.....	98	8.1 函数的概念	142
5.4.1 break 语句.....	98	8.2 函数定义	143
5.4.2 continue 语句	100	8.3 函数声明	145
本章小结	101	8.4 函数调用	147
习题	101	8.5 变量的作用域类型.....	149
第6章 宏和编译预处理.....	102	8.5.1 局部变量.....	149
6.1 宏定义	103	8.5.2 全局变量.....	151

8.6 变量的存储类型	152	9.5.2 new 和 delete 运算符	193
8.6.1 动态存储变量	152	9.5.3 指针与数组	194
8.6.2 静态存储变量	153	9.6 动态内存分配的应用	198
8.7 函数返回值	153	9.6.1 应用举例 1	198
8.8 默认函数参数	155	9.6.2 应用举例 2	199
8.9 内联函数	158	9.7 const 指针	204
8.10 函数重载	160	9.8 指针作为函数参数	208
8.11 作用域	163	9.9 指针函数	213
8.11.1 局部作用域	163	9.10 函数指针	215
8.11.2 函数作用域	164	9.11 指针数组	219
8.11.3 函数原型作用域	165	9.12 指向指针的指针	223
8.12 可见性与生命期	165	9.13 常见的内存错误及其对策	226
8.12.1 可见性	165	9.14 引用的定义	227
8.12.2 生命期	167	9.15 使用引用访问数据	232
8.12.3 补充说明	168	9.16 引用与指针对比	235
8.13 综合应用举例	168	9.17 引用做函数的参数	236
8.14 递归函数	170	9.18 应用举例 3	239
8.14.1 递归函数举例	170	9.19 返回引用	240
8.14.2 递归调用过程分析	171	9.20 函数调用作为左值	244
8.14.3 递归程序设计方法	172	9.21 const 限定的引用	246
8.15 程序文件结构	174	9.22 返回堆中变量的引用	248
8.15.1 头文件	174	本章小结	250
8.15.2 文件作用域	176	习题	250
8.15.3 多文件结构	176		
8.15.4 外部存储类型	177	第 10 章 结构、联合、枚举	252
本章小结	179	10.1 自定义数据类型概述	252
习题	179	10.2 结构的定义	253
第 9 章 指针和引用	180	10.3 结构初始化	255
9.1 指针的概念	180	10.4 访问结构成员	257
9.2 指针声明和赋值	182	10.5 结构与数组	258
9.3 通过指针访问数据	185	10.6 结构与指针	264
9.4 指针运算	187	10.7 结构与引用	266
9.5 动态内存分配	190	10.8 在函数中使用结构	267
9.5.1 malloc() 和 free() 函数	191	10.9 结构的复杂形式	273
		10.10 链表	275

10.11 联合	279
10.12 枚举	285
本章小结	294
习题	294

第 11 章 类与对象

11.1 抽象概述	296
11.2 类的概念	297
11.3 类的定义	298
11.3.1 类与结构	298
11.3.2 类的声明	299
11.3.3 类成员的访问控制	300
11.3.4 数据成员	303
11.3.5 成员函数	304
11.3.6 重载成员函数	306
11.3.7 类定义的注意事项	307
11.3.8 类声明和类定义	308
11.4 对象	309
11.4.1 类与对象的区别和联系	309
11.4.2 对象的声明	309
11.4.3 访问数据成员	310
11.4.4 调用成员函数	310
11.5 综合应用	313
11.6 构造函数	314
11.6.1 为何需要构造函数	314
11.6.2 构造函数的定义	316
11.6.3 带参数构造函数	317
11.6.4 默认构造函数	318
11.6.5 重载构造函数	319
11.7 类对象成员的初始化	320
11.8 析构函数	323
11.8.1 为何需要析构函数	323
11.8.2 析构函数的定义	324
11.8.3 何时需要使用析构函数	325
11.9 堆栈和内存分配	325

11.9.1 内存管理概述	325
11.9.2 变量与对象的空间分配 时机与初始化	327

11.9.3 为什么使用 new/delete 操作符	328
--------------------------------------	-----

11.10 拷贝构造函数	329
11.10.1 程序出错的原因分析	329
11.10.2 拷贝构造函数	332
11.10.3 默认拷贝构造函数	334
11.10.4 浅拷贝与深拷贝	334
11.11 临时对象和无名对象	337
11.11.1 临时对象	337
11.11.2 无名对象	338
11.12 const 成员	339
本章小结	340
习题	341

第 12 章 静态成员与友元

12.1 静态成员	342
12.1.1 为何需要静态成员	342
12.1.2 静态成员变量	343
12.1.3 静态成员函数	345
12.2 友元	346
12.2.1 为何需要友元	346
12.2.2 友元函数	347
12.2.3 友元类	348

第 13 章 继承与多态

13.1 继承与派生的概念	352
13.2 继承的实现方式	354
13.3 继承类的构造与析构	355
13.3.1 继承类的构造	355
13.3.2 构造函数的参数传递	357
13.4 基类访问控制	360
13.5 多态与虚函数	362
13.5.1 为什么使用虚函数	363

13.5.2 虚函数.....	364	15.3.1 类模板的定义.....	401
13.5.3 重载、隐藏与覆盖.....	365	15.3.2 使用类模板.....	403
13.5.4 虚函数的限制.....	368	15.4 综合应用.....	404
13.6 多继承.....	370	本章小结.....	406
13.6.1 多继承的实现.....	371	第 16 章 标准模板库	407
13.6.2 多继承的二义性.....	371	16.1 标准模板库的基本组成.....	407
13.6.3 多继承的构造顺序.....	373	16.2 标准命名空间.....	408
13.7 虚基类.....	374	16.3 容器.....	409
13.7.1 虚基类的定义.....	375	16.3.1 顺序容器.....	409
13.7.2 虚基类的构造函数和 初始化.....	376	16.3.2 关系式容器.....	411
本章小结.....	376	16.4 迭代器.....	413
习题.....	377	16.5 算法.....	415
第 14 章 运算符重载	378	本章小结.....	418
14.1 为何要重载运算符.....	378	第 17 章 I/O 流	419
14.2 运算符重载的实现.....	379	17.1 流的概念.....	419
14.2.1 成员函数运算符重载.....	380	17.2 I/O 标准流类.....	421
14.2.2 友元函数运算符重载.....	382	17.2.1 标准流的设备名.....	421
14.3 单目运算符和双目运算符.....	383	17.2.2 原理.....	421
14.4 引用返回和值返回.....	386	17.3 输入输出的格式控制.....	423
14.5 常见的运算符重载.....	391	17.3.1 用 ios 类的成员函数进行 格式控制.....	423
14.5.1 赋值与比较运算符.....	391	17.3.2 使用格式控制符进行 格式控制.....	427
14.5.2 递增与递减运算符.....	392	17.4 输入输出运算符的重载.....	429
14.5.3 数组存取标识符.....	393	17.4.1 重载输出运算符.....	429
本章小结.....	395	17.4.2 重载输入运算符.....	430
习题.....	395	17.5 文件流.....	431
第 15 章 模板	396	17.5.1 打开文件.....	432
15.1 为何需要模板.....	396	17.5.2 关闭文件.....	434
15.2 函数模板.....	397	17.5.3 写入文件.....	435
15.2.1 函数模板的概念.....	397	17.5.4 读取文件.....	435
15.2.2 函数模板的定义.....	398	17.5.5 常用操作和状态检测.....	436
15.2.3 函数模板的实例化.....	399	17.5.6 读写随机文件.....	438
15.2.4 重载模板函数.....	399	17.5.7 二进制文件的处理.....	440
15.3 类模板.....	401		

本章小结	442	18.4 构造和析构中的异常抛出	448
习题	442	18.5 异常规格说明	449
第 18 章 异常处理	443	18.6 标准异常	451
18.1 传统的错误处理方式	443	本章小结	452
18.2 异常处理的思想	445	附录 A 程序的写作风格	453
18.3 异常处理的实现方式	447		



第 1 章 概观程序设计

本章内容:

- 计算机程序发展历程。
- 程序设计思想变迁史。

重点:

面向对象编程思想。

目的:

了解程序设计发展史。

1.1 程序设计发展历程

1.1.1 什么是计算机程序

自 1946 年世界上第一台电子计算机问世以来, 计算机科学及其应用的发展十分迅猛, 计算机被广泛地应用于人类生产、生活的各个领域, 推动了社会的进步与发展。特别是随着国际互联网(Internet)日益深入千家万户, 传统的信息收集、传输及交换方式正被革命性地改变, 人类已经难以摆脱对计算机的依赖, 计算机已将人类带入了一个新的时代——信息时代。

一台计算机是由硬件系统和软件系统两大部分构成的。硬件是计算机的物质基础; 而软件可以说是计算机的灵魂, 没有软件, 计算机只是一台“裸机”, 什么也不能干, 有了软件, 才能灵活起来, 成为一台真正的“电脑”。而所有的软件, 都是使用计算机语言编写出来的。

软件程序是控制计算机完成特定功能的一组有序指令的集合, 编写程序所使用的语言称为程序设计语言。电脑所做的每一次动作, 每一个步骤, 都是按照已经用计算机语言编好的程序来执行的。程序是计算机要执行的动作的集合, 而程序全部都是用程序设计语言来编写的。所以人们要控制计算机一定要通过计算机语言向计算机发出命令。程序设计语

言经历了机器语言、汇编语言到高级语言的多个阶段。

随着计算机技术的不断进步和计算机图形技术的发展，计算机程序提供了对现实世界越来越逼真的模拟，在这种情况下一种新的计算机程序出现了，这种程序的目的是不再是为了计算复杂的弹道数据，或是模拟宇宙的演化，它只是为了一个简单的目标：娱乐。就这样，电子游戏产生了，同时也产生了游戏程序员。

游戏程序员的程序设计能力决定着他能为游戏策划实现哪些功能，能做出什么样的游戏，可以说程序员的水平决定了游戏的成败。对于游戏程序员而言，掌握一门高级语言及其基本的编程技能是必需的。不仅如此，作为专业技术人员，除了掌握本专业系统的基础知识外，科学精神的培养、思维方法的锻炼、严谨踏实的科研作风的养成，以及分析问题、解决问题的能力训练，都是日后工作的基础。学习计算机语言，正是一种十分有益的训练方式，而语言本身又是与计算机进行交互的有力工具。

1.1.2 计算机程序语言的发展历史

计算机程序设计发展的历史，也就是计算机程序设计语言的发展历史，它经历了从机器语言、汇编语言到高级语言的历程。

1. 机器语言

电子计算机所使用的是由“0”和“1”组成的二进制数，二进制是计算机的语言基础。在计算机发明之初，计算机所能识别的语言只有机器语言，即由0和1构成的代码。所以人们只能放弃自己的自然语言，用计算机的语言去命令计算机干这干那儿，也就是写出一串串由“0”和“1”组成的指令序列交由计算机执行，这种语言就是机器语言。

使用机器语言是十分痛苦的，特别是在程序有错需要修改时更是如此。而且，由于每台计算机的指令系统往往各不相同，所以，在一台计算机上执行的程序，要想在另一台计算机上执行，必须另编程序，这就造成了重复工作。但由于使用的是针对特定型号计算机的语言，且不需要经过翻译、编译等过程，而是直接执行的，故而它的运算效率是所有语言中最高的。机器语言，是第一代计算机语言。

2. 汇编语言

为了减轻使用机器语言编程的痛苦，提高程序开发的效率，人们进行了一种有益的改进：用一些简洁的英文字母、符号串来替代一个特定指令的二进制串，比如，用“ADD”代表加法，“MOV”代表数据传递等。这样一来，人们就比较容易读懂并理解程序在干什么，纠错及维护都变得方便了，这种程序设计语言就称为汇编语言，即第二代计算机语言。然而计算机是不认识这些符号的，这就需要一个专门的程序，专门负责将这些符号翻译成二进制数的机器语言，这种翻译程序被称为汇编程序。

汇编语言同样十分依赖于机器硬件，移植性不好，但效率仍十分高，针对计算机特定硬件而编制的汇编语言程序，能准确发挥计算机硬件的功能和特长，程序精练而质量高，所以至今仍是一种常用且强有力的软件开发工具。

3. 高级语言

从最初与计算机交流的痛苦经历中，人们意识到，应该设计一种这样的语言，这种语言接近于数学语言或人的自然语言，同时又不依赖于计算机硬件，编出的程序能在所有机器上通用。经过不懈的努力，在1954年，第一个完全脱离机器硬件的高级语言——FORTRAN问世了，40多年来，共有几百种高级语言出现，有重要意义的有几十种，影响较大、使用较普遍的有FORTRAN、ALGOL、COBOL、BASIC、LISP、SNOBOL、PL/1、Pascal、C、PROLOG、Ada、C++、Delphi和Java等。

高级语言的发展也经历了从早期语言到结构化程序设计语言，从面向过程到面向对象程序语言的过程。相应的，软件的开发也由最初的个体手工作坊式的封闭式生产，发展为产业化、流水线式的工业化生产。

过去的软件生产基本上是以个人为单位进行开发，缺乏科学规范的系统规划与测试、评估标准。当这样的工作方式应用于大型软件的开发时，恶果是大批耗费巨资建立起来的软件系统，由于含有错误而无法使用，甚至带来巨大损失，软件给人的感觉是越来越不可靠，以致几乎没有不出错的软件。这一切，极大地震动了计算机界，史称“软件危机”。人们认识到：大型程序的编制不同于写小程序，它应该是一项新的技术，应该像处理工程一样处理软件研制的全过程。程序的设计应易于保证正确性，也便于验证正确性。1969年，荷兰计算机科学家迪克斯特拉(E.W.Dijkstra)提出了结构化程序设计方法；1970年，第一个结构化程序设计语言——Pascal语言出现，标志着结构化程序设计时期的开始。

从20世纪80年代初开始，在软件设计思想上又产生了一次革命，其成果就是面向对象的程序设计思想。在此之前的高级语言，几乎都是面向过程的，程序的执行如同流水线似的，在一个模块被执行完成前，人们不能干别的事，也无法动态地改变程序的执行方向，这和人们日常处理事物的方式是不一致的。对人而言，希望发生一件事就处理一件事。也就是说，不是面向过程，而应是面向具体的应用功能，也就是对象(Object)。其方法就是软件的集成化，如同硬件的集成电路一样，生产一些通用的、封装紧密的功能模块，称之为软件集成块，它与具体应用无关，但能相互组合，完成具体的应用功能，同时又能重复使用。对使用者来说，只需关心它的接口(输入量、输出量)及能实现的功能，至于是如何实现的，那是它内部的事，使用者完全不用关心，C++、Delphi就是这种语言的典型代表。

随着技术的不断进步，高级语言的下一个发展目标是面向应用，也就是说，只需要告诉程序你要干什么，程序就能自动生成算法，自动进行处理，这就是非过程化的程序语言。

1.2 程序设计思想

1.2.1 结构化程序设计思想

结构化程序设计(Structured Programming, SP)的思想是由荷兰学者 E.W.Dijkstra 提出的,它规定了一套方法,使程序具有合理的结构,以保证和验证程序的正确性。这种方法要求程序设计者不能随心所欲地编写程序,而是要按照一定的结构形式来设计和编写程序。它的一个重要目的是使程序具有良好的结构,使程序易于设计,易于理解,易于调试修改,以提高设计和维护程序工作的效率。

结构化程序设计是 E.W.Dijkstra 等人在研究人的智力局限性随着程序规模的增大而表现出不适应的特点之后,于 1969 年提出的一种程序设计方法,这是一种在解决复杂问题时避免混乱的技术。它提出了把程序结构规范化的主张,要求对复杂问题的求解过程应按人们大脑容易理解的方式进行组织,而不是强迫人们的大脑去接受难以忍受的冲击。具体来说,结构化程序设计的思想包括以下三方面的内容。

- 程序由一些基本结构组成。任何一个大型的程序都由三种基本结构所组成,由这些基本结构顺序地构成一个结构化的程序。这三种基本结构为:①顺序结构,如图 1-1 所示;②选择结构(亦称分支结构),如图 1-2 所示;③循环结构,如图 1-3 所示。

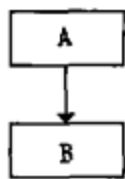


图 1-1 顺序结构

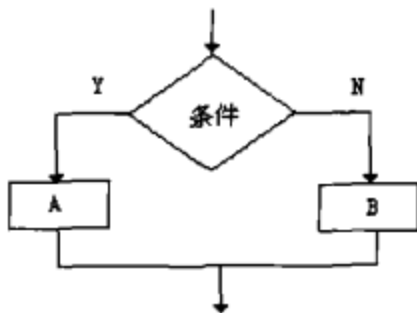


图 1-2 选择结构

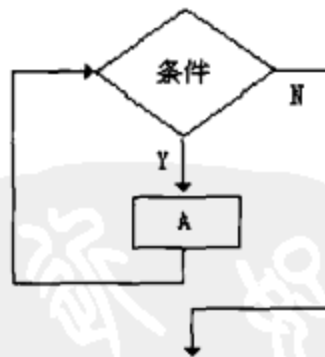


图 1-3 循环结构

同时结构化定理还进一步表明,任何一个复杂问题的程序设计都可以用顺序、选择和循环这三种基本结构组成,且它们都具有以下特点:只有一个入口;只有一个出口;结构中无死循环。程序中三种基本结构之间形成顺序执行关系。

- 一个大型程序应按功能分割成一些功能模块,并把这些模块按层次关系进行组织。
- 在程序设计时应采用自顶向下、逐步细化的实施方法。

用八个字来描述结构化程序设计的基本思想就是：自顶向下，逐步求精。

“自顶向下”是将大而复杂的问题划分为多个小问题，找出问题的关键、重点所在，然后用精确的思维定性、定量地去描述问题。

“逐步求精”是将现实世界的问题经抽象化处理转化为逻辑空间或求解空间的问题；复杂问题经抽象化处理变为相对简单的问题。经若干步抽象(精化)处理后，最后到求解域中的只是比较简单的编程问题。

在这个“自顶向下、逐步求精”的细化过程中，要对系统功能逐层分解出许多易于理解和实现的、逻辑上相对独立的子功能，形成一棵功能树。在程序实现时，功能树上的每一个结点对应一个函数。一个C程序是由一到多个函数组成；一个C++程序不但可以包含多个函数，而且还可以包含多个类，而一个类就包含一到多个成员函数。

按结构化程序设计方法设计出的程序优点是：结构良好、各模块间的关系清晰简单、每一模块内都由基本单元组成。这样设计出的程序清晰易读、可理解性好、容易设计、容易验证其正确性，也容易维护。同时，由于采用了“自顶向下、逐步细化”的实施方法，能有效地组织人们的智力，有利于软件的工程化开发。

1.2.2 面向对象程序设计思想

随着计算机技术的不断发展，其软硬件之间的差距越来越大，这就造成了计算机发展的不均衡。当系统较为复杂时，常规的软件工具、技术和概念已不足以应付，从而使软件开发陷入了困境，即所谓的“软件危机”。尽管软、硬件发展的这种差距自计算机出现以来始终存在，但进入20世纪90年代后这种差距更加明显。在这一背景下，面向对象(Object-Oriented Programming, OOP)程序设计技术逐渐兴起，而C++这种最为优秀的面向对象语言也应运而生，随着它的不断完善，其逐步进入实用阶段并受到广大软件开发者的青睐，吸引了众多的人士去研究和用它。而面向对象的思想也在软件工程、人工智能等领域得到了十分广泛的应用。面向对象的程序设计语言的出现是计算机软件产业的一次革命。

1. 面向对象程序方法的基本思想及对象的含义

面向对象程序设计方法的基本思想是：认为世界由各种对象组成，任何事物都是对象，是某个对象类的实例；复杂的对象可以由比较简单的对象以某种方式组成。按照这个观点，整个世界也可以从一些最原始的对象开始，经过层层组合而成。因此，可以说整个世界就是一个最复杂的对象。

在面向对象程序设计中，对象是系统中的基本运行实体，是有特殊属性(数据)和行为

方式(方法)的实体。即对象由两个元素构成：一组包含数据的属性和允许对属性中包含的数据进行操作的方法。也可以说，对象是将某些数据代码和对该数据的操作代码封装起来的模块。

对象包含了数据和方法，每个对象就是一个微小的程序。由于其他对象不能直接操纵该对象的私有数据，只有对象自身的方法才能得到它，这就使对象具有很强的独立性，因此可把对象当作软件的基本组件。在面向对象的程序设计中，可使用若干对象来建立所需要的各种复杂的应用软件，即通过对象组合，创建具体的应用。

因此，把所有对象都划分成各种对象类，对每个对象类都定义一组方法。所谓方法实际上是允许施加于该类对象上的各种操作。对象和传统的数据有本质区别，它不是被动地等待对其执行某种操作；相反，它是进行处理的主体，必须发送消息请求对象执行它的某个操作，处理它的私有数据，而不能从外界直接对它的私有数据进行操作。对象之间除了互相传递消息的联系之外再没有其他联系。一切局部对象的信息和实现方法，都被封装于相应的对象类的定义之中，在外界是不可见的。

2. 面向对象程序设计语言的特性

面向对象程序设计语言中最为精彩的是它的三大特性：封装性(Encapsulation)、继承性(Inheritance)和多态性(Polymorphism)。

1) 封装性

封装就是将对象的属性和方法封装到具有适当定义接口的容器中。对象接口提供的方法和属性应使对象能够如期使用。封装的功能取决于两个重要概念：模块化和信息隐藏。模块化是对象的自给自足的特性，它不会访问定义接口以外的其他对象。信息隐藏是指将对象暴露的信息限制在对象接口的使用所必需的范围内，删除对象中仅供对象内部操作的信息。

封装是一种信息隐蔽技术，用户只能见到对象封装界面上的信息，对象内部对用户是隐蔽的。封装的目的在于将对象的使用者和设计者分开，使用者不必知道行为实现的细节，只需用设计者提供的消息来访问该对象即可。

2) 继承性

对象类按照类、子类与父类的关系构成了一个层次结构的系统。在这种层次结构中，上层对象类所具有的性质可以被下层对象类继承，除非在下层对象类中又对相应的属性重新作了描述，这时将以新属性为准。也就是说，低层的属性将屏蔽高层的同名属性，这种特性称为对象类之间的继承。继承是实现软件重用的重要机制。

继承是从一种对象类型构造另一种对象类型的一个主要方法。利用继承性，可以在已

经定义的对象类型基础上创建更复杂、更专业的对象类型，只要加进所需的属性和方法，将新对象与上级对象区别开来即可。继承性是自动的共享类、子类和对象中的方法和数据的机制，合理使用继承可以减少很多的重复劳动。如果某个类实现了一个特别的功能，那么它的派生类就可以重复使用这些功能，而不需要再重新编程。

3) 多态性

所谓多态是指一个名字可具有多种语义，多态引用表示可引用多个类的实例。多态可为一种对象类定义一种方法的多种实现方案，这些方法是通过类型和可接收的参数来区分而加以使用的。

多态性可使公共的信息传送给基类对象及所有的派生类对象，允许每一个基类对象按适合于其定义的方式响应信息格式。

多态性有时也指方法的重载。方法的重载是指同一个方法(即函数)名在上下文中有不同的含义，是实现重载的类以统一的方式处理不同数据类型的一种手段。它是静态的，这是因为在实现类编写方法之前需要考虑将要遇到的所有数据类型。子类在动态运行时提供了更丰富的多态性。

对象的以上这些特性使它具有很强的可重用性。用户在开发应用程序时，可以调用系统中的各种对象，作为自己应用程序的基本组件，这样就使新增代码明显减少，而且增加了程序的可靠性和可维护性。

3. 面向对象程序设计存在的问题

面向对象程序设计技术确实提供了与真实世界较为接近的模型，为提高软件开发生产率提供了诱人的前景，然而面向对象程序设计技术不是万能的，作为一种新兴的技术在实现上还存在着某些缺陷，归纳起来主要有以下几个方面的问题。

1) 运行效率问题

面向对象程序设计技术是针对当前的软件开发危机而产生的，它在提高编程效率方面所起的作用是毋庸置疑的，但是在运行时，用面向对象技术开发的程序通常较用过程化方法开发的程序的效率要低，虽然随着 CPU 速度的提高、内存容量的增加，对一般规模的面向对象系统其运行速度用户是可以接受的，但当系统规模较大时，这一问题不容忽视。

2) 类库的化简问题

面向对象程序设计语言通常都提供了一个具有丰富功能的类库，用户可以有选择地使用它们进行程序开发，从而缩短软件开发生命周期。要成为一名高效的面向对象程序员必须能熟练地运用类库，掌握类库中各个类提供的功能，但由于类库都过于庞大，程序员对

它们的掌握要有一个时间过程，从普及、推广的角度，类库应在保证其功能完备的基础上进行相应的缩减。

3) 类库的可靠性问题

虽然类库中提供的类都是经过精心设计和测试过的，但如此庞大的系统谁也无法保证类库中的每个类在各种环境中都能百分之百的正确。如果应用程序中使用了类库中某个存在问题的类，当经过几层继承后错误才显现出来，这时程序员对此将束手无策，有可能要推翻原来的全部工作，虽然这种情况出现的概率很低，但国外确曾有过这方面的报道。

以上谈及的这些问题，不应该成为阻碍面向对象技术发展的障碍，随着软、硬件技术的发展及面向对象技术自身的完善，必将会给软件开发技术、观念带来革命性的转变，使软件生产进入一个新的时代。

本章小结

本章概要地介绍了程序设计的基本概念以及程序设计的发展历程，明确指出了目前流行的程序设计思想为面向对象编程思想。



第 2 章 开发环境简介

本章内容:

- Visual C++ .NET 的菜单组成。
- Visual C++ .NET 开发管理方式。
- Visual C++ .NET 支持的资源。
- Linux 下的开发环境配置及程序开发方法。

重点:

解决方案管理与代码编写。

目的:

掌握利用各种开发环境编写 C++ 程序的流程。

2.1 Visual Studio .NET 集成开发环境

Visual Studio .NET 2008 是 Microsoft 的新一代开发工具, 用于构建和部署功能强大而安全的 Windows 平台应用软件, 它是 Microsoft Windows 环境下的一个支持可视化编程的集成开发环境(Integrated Development Environment, IDE), 通常称为开发平台。集成开发环境是一个集成程序编译器、调试工具和建立应用程序工具的主体。

作为 Visual Studio .NET 2008 的核心组件, Visual C++ .NET 2008 集成了开发 C++ 和 C 应用程序所需要的所有工具和 C++ 编译器, 是一个完善的 C++ 程序开发平台。Visual C++ .NET 2008 可以用来管理项目、生成和编辑程序源文件、设计程序资源(如菜单、对话框与图标等)和产生程序的基本框架及一些源代码, 可以在 Visual C++ .NET 2008 中直接建立和调试程序, 还可以检查和管理程序符号与 C++ 类, 访问 Visual C++ .NET 2008 联机帮助, 使用 Help 菜单命令等。

微软公司在推出这一款继承开发工具时, 为了达到更高的效率, 将 4 种语言: Visual Basic、Visual C#、Visual C++ 和 Visual J# 集成到了统一开发环境中。如果用户只需要使用 Visual C++ 部分的内容, 在进行 Visual Studio .NET 2008 的安装时可以选择只安装 Visual

C++ .NET 2008 组件, 对于选择安装全部语言组件的 Visual Studio .NET 2008 的用户来说, 在菜单中可以通过选择操作, 执行不同语言的编译任务。

由于 Visual C++ .NET 2008 是一个可视化的开发工具, 所以开发过程多数表现为单击鼠标按钮和拖放图形化对象以及设置对象的属性、行为的过程。这种可视化的编程方法易学易用, 所见即所得, 可以提高工作效率。

当启动 Visual C++ .NET 2008 后, 可以看到它提供的 IDE 环境如图 2-1 所示。

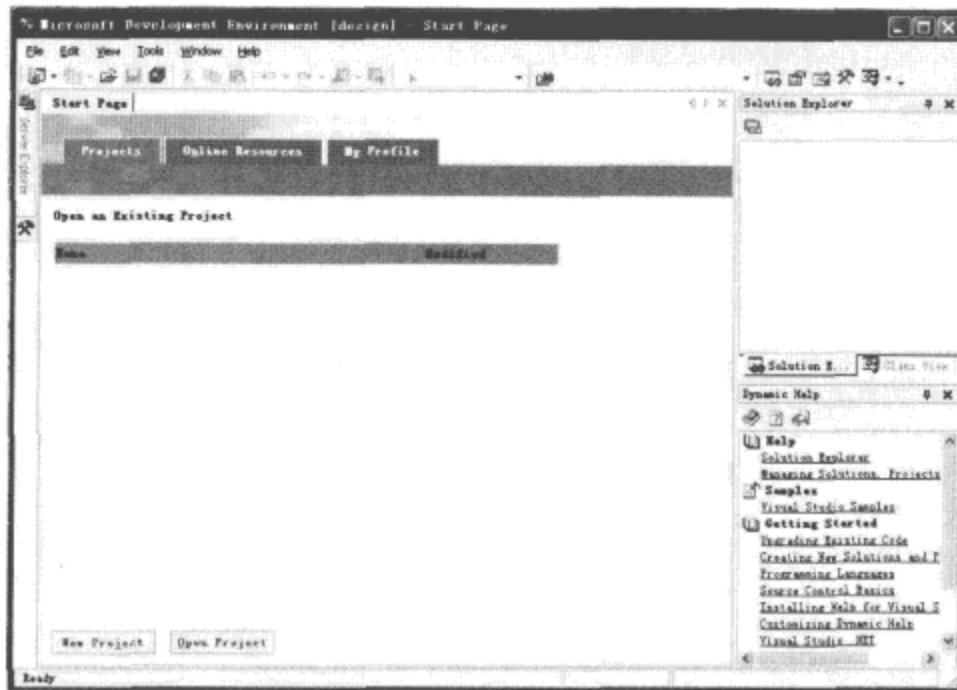


图 2-1 Visual C++ .NET 2008 集成开发环境

从图 2-1 中可以看到, Visual C++ .NET 2008 的 IDE 界面和传统的 Windows 程序界面一样, 包含菜单栏、工具栏以及状态栏等。但 IDE 界面还有重要的三大部分: Solution Explorer(解决方案资源管理器)窗口、Workspace(工作区)和 Output(输出)窗口。这些基本界面元素相互配合, 形成了具有 Visual C++ .NET 2008 特色的 IDE 环境。下面简单介绍一下创建项目的步骤。

2.1.1 创建项目

菜单栏提供了 Visual C++ .NET 2008 几乎所有的命令。通过这些命令, 可以访问到 Visual C++ .NET 2008 的大部分功能。除了提供标准的 File、Edit、View、Window 和 Help 菜单外, Visual C++ .NET 2008 还提供了编辑专用的功能菜单, 如 Project、Tools、Build 和 Debug 等。

File(文件)菜单的各个命令主要用于完成文件的建立、保存、打开、关闭以及打印等工作。其中最常用的命令是 New(新建)。

在 File 菜单中使用鼠标选择 New 命令会出现 3 个子选项, 如图 2-2 所示, 3 个子选项分别用于实现如下三种功能。



图 2-2 New 命令菜单

- Project(项目): 新建项目, Visual C++ .NET 2008 会为用户提供部分类型项目的框架向导, 如图 2-3 所示。
- File(文件): 新建文件, 可以新建多个类型的文件。
- Blank Solution(空白解决方案): 新建空白解决方案。

选择 New→Project 命令后将打开 New Project 对话框, 如图 2-3 所示。

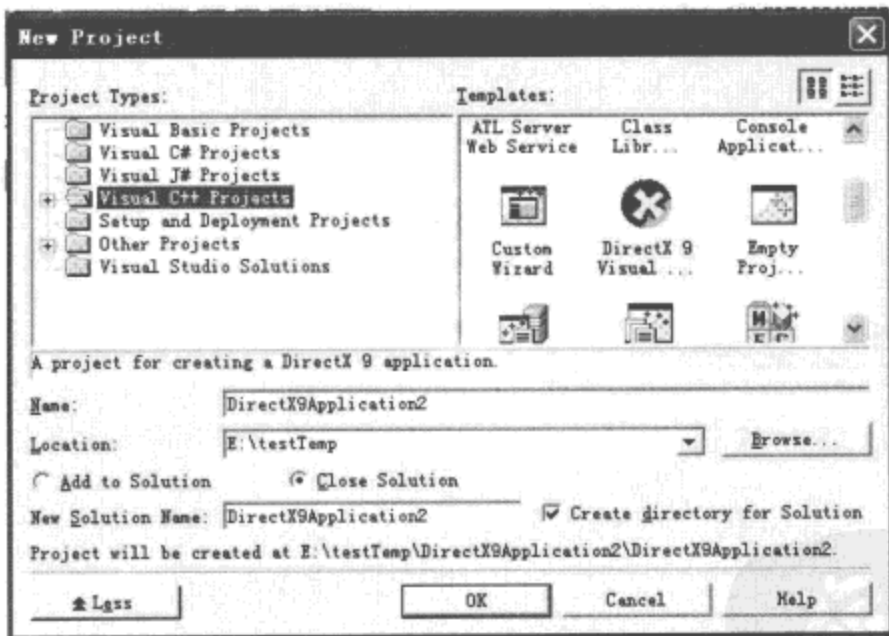


图 2-3 New Project 对话框

在 New Project 对话框的 Templates 列表框中选择需要创建的项目类型, 然后在 Name 文本框中输入项目的名字和设置 Location 文本框中的项目存储路径, 单击 OK 按钮就可以新建需要的项目文件, 集成开发环境将会为用户生成对应类型程序的基本框架。可以创建的项目类型有: ATL COM AppWizard(Active Template Library 应用程序向导)、Custom AppWizard(自定义的应用程序向导)、MFC Application(MFC 应用程序)、MFC DLL(MFC 动态链接库)、Win32 Project(Win32 应用程序)和 Win32 Console Application(Win32 控制台应用程序)等。

创建完新项目后, 集成开发环境将自动生成项目框架代码, 并自动为项目生成对应的解决方案。解决方案是项目组织管理的单位, 一个解决方案可以包括对应的多个项目。

2.1.2 创建文件

选择 New→File 命令后,可以进入 New File 对话框(如图 2-4 所示),用户可以选择新建的文件有:Bitmap File(位图文件)、C/C++Header File(C/C++头文件)、C++Source File(C++源文件)、Cursor File(光标文件)、HTML Page(HTML 页文件)、Icon File(图标文件)和 Text File(文本文件)等。

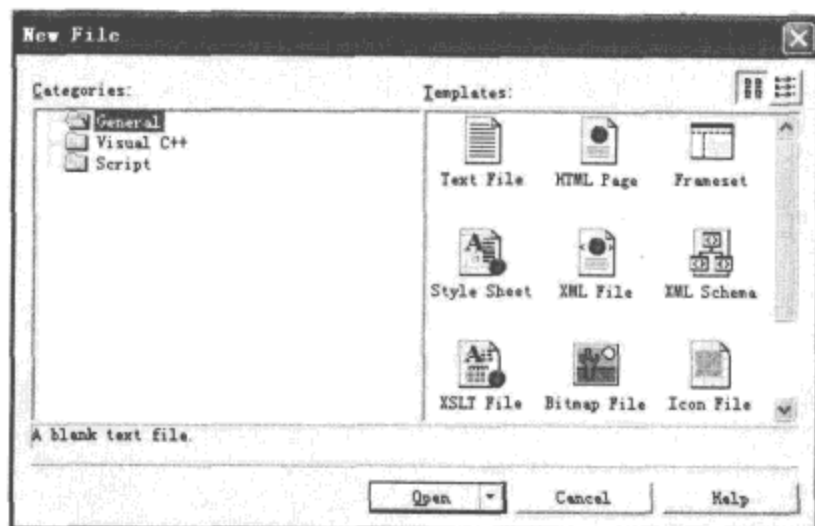


图 2-4 New File 对话框

2.1.3 项目属性设置

创建完项目后,需要对项目的属性进行必要的设置,在创建大型项目的时候更是如此。

在 Visual Studio 主窗口左边的解决方案中,右键单击项目名称,在弹出的快捷菜单中选择“属性”命令,在弹出的对话框中可以进行多种属性设置,如图 2-5 所示。

在实际工作中,最需要经常设置的属性是附加包含目录、附加库目录、代码生成属性这三项。

展开 C/C++树,在其右侧列表中,选择“附加包含目录”选项,在弹出的窗口中,选择需要的附加头文件所在的目录即可,这里可以多选,将来程序在编译期间,会按照选择的先后次序进行头文件的查找。

同样,在 C/C++子树中,“代码生成”选项也很有用。当使用不同类型的函数库时,应该选择对应类型的调试项,例如若使用了单线程调试库,则这里应该选择 MTD 项,其他类似。

展开“链接器”树,在该树中,常用的设置项目是常规选项中的“附加库目录”选项,设置原理类似于附加包含目录,而在“输入”项目中,可以设置用户程序中使用到的第三

- Clean Solution(清理解决方案)命令: 清理解决方案, 删除原来生成的编译文件。
- Build Project(生成项目)命令: 和当前处于选中状态的项目对应, 用于编译当前选中的项目。
- Rebuild Project(重新生成项目)命令: 和当前处于选中状态的项目对应, 用于清理并重新编译当前选中项目。
- Compile(编译)命令: 编译当前文件。

2.1.5 调试

Debug 菜单用于对完成编译的程序进行调试工作, 该菜单中几个常用命令的介绍如下。

- Start(启动): 在调试状态启动并开始调试程序。
- Start without debug(启动但不调试): 启动程序但是不处于调试状态。
- New breakpoint(新断点): 设置新的断点, 用于进行程序调试。
- Clear All BreakPoint(清除所有断点): 清除用户设置的所有断点。
- Step Into(执行至下一步): 单步执行到函数内部。
- Step over(单步执行): 单步执行, 只在当前文件下执行。

2.1.6 辅助工具

1. Tools 菜单

Tools 菜单中的命令用于浏览用户程序中定义的符号、定制菜单与工具栏, 激活常用的工具或更改选项和变量设置等。Tools 菜单包括以下几个常用命令。

1) Error Lookup 命令

可以使用该命令在输入值的基础上恢复一个系统错误消息或模块错误消息。如果从调试器或其他支持 OLE(Object Linking and Embedding, 对象链接与嵌入)的应用程序上拖动了一个十六进制数或十进制数, 它将自动恢复错误消息文本。

2) ActiveX Control Test Container 命令

该命令启动一个随 Visual C++ 一起发行的测试容器(Test Container)应用程序。此程序是一个用于测试 ActiveX 控件的 ActiveX 控件容器。测试容器允许开发者通过改变控件的属性、使用其方法、激活其事件来测试其功能。

3) OLE/COM Object Viewer 命令

该命令显示安装在计算机上的 ActiveX 控件、OLE 对象以及它们支持的界面, 利用它也可以编辑注册表和检查类型库。

4) Spy++命令

该命令启动一个 Win32 实用程序，可用于显示系统对象间相互关系的图形化表示；搜索指定的窗口、线程、进程或消息；查看选定对象的属性；从视图中直接选择一个窗口、线程、进程或消息。

5) Customize 命令

该命令用于定制集成开发环境的界面，选择该命令将弹出 Customize 对话框，如图 2-7 所示，它可以对命令、工具栏、工具菜单、键盘快捷键和宏进行定制。

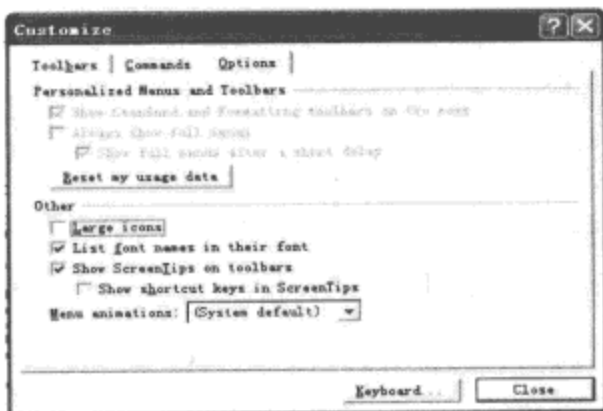


图 2-7 Customize 对话框

6) Macro 命令

该命令用于创建、编辑和运行宏。所谓宏就是用 Visual Basic Scripting Edition 语言编写的程序。正确运用宏可以简化在集成开发环境中的工作。因为在一个宏中可以结合几个命令，从而加快程序编制，或自动执行一系列复杂任务。

7) Create GUID 命令

该命令用于创建唯一的全局标识符 GUID。单击该命令后出现的创建对话框如图 2-8 所示。

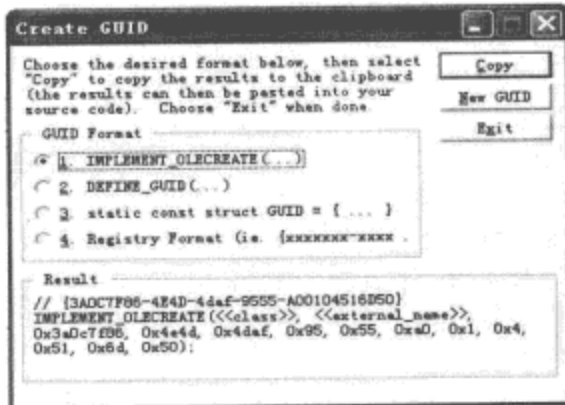


图 2-8 Create GUID 对话框

2. 工具栏

工具栏提供了对常用命令快速执行的各种手段,使用起来更加方便快捷,不必在一层一层的菜单中查找。一般来说,工具栏所提供的功能,在菜单中都有相应的命令,工具栏的命令是菜单命令的一个子集。

Visual C++ .NET 2008 提供了 10 个不同的工具栏,分别是 Standard、Build、Build Minibar、ATL、Resource、Edit、Debug、Browse、Data6BSe 和 WizardBar 工具栏。默认时,Standard、Build Minibar 和 WizardBar 工具栏显示在主窗口中。

1) 定位或浮动工具栏

Visual C++ .NET 2008 拥有标准工具栏(在 Visual C++ .NET 2008 标题栏下面可以找到)。这些工具栏可以处于固定状态或者浮动状态。

在固定状态下,工具栏被固定在应用程序窗口的任意一个边上,当工具栏被固定时就不能改变其大小了。

在浮动状态下,工具栏具有一个细的标题栏,并且可以出现在屏幕的任何地方,浮动工具栏经常处于所有窗口的最前端。当工具栏处于浮动状态时,我们可以改变它的位置和大小。

只要将固定的工具栏用鼠标拖曳到窗口中你所希望的位置,固定工具栏就会变成浮动的工具栏;双击浮动工具栏的标题栏,浮动工具栏的标题栏将返回到以前的位置变成固定工具栏。

为了使用工具栏,必须使工具栏出现在 Visual C++ .NET 2008 的主窗口中,方法是移动鼠标指向工具栏的位置,然后右击,屏幕上会弹出一个快捷菜单(如图 2-9 所示),单击要使用的工具栏,使其前面出现“√”,则相应的工具栏就会出现在主窗口中。同样的方法可以实现显示/隐藏工具栏。

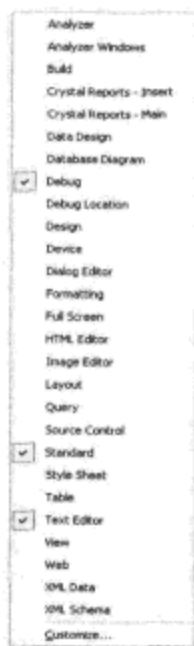


图 2-9 工具栏选择快捷菜单

2) 常用工具栏

下面介绍几种常用工具栏所含的各种工具的作用、用法及工具栏的定制过程。

(1) Standard 工具栏。

Standard 工具栏含有 15 个工具,主要用于建立项目工作区及项目。从左至右各按钮的具体功能如表 2-1 所示。

表 2-1 Standard 工具栏

工 具	功 能
New Text File	创建新的文本文件
Open	打开已有的文档
Save	保存文档
Save All	保存所有打开的文件
Cut	剪切选定的内容到剪贴板中
Copy	复制选定的内容到剪贴板中
Paste	在当前插入点处插入剪贴板中的内容
Undo	取消最后的操作
Redo	重复先前的操作
Workspace	显示或隐藏工作区窗口
Output	显示或者隐藏输出窗口
Windows List	管理当前打开的窗口
Find in Files	在多个文件中搜索字符串
Find	激活查找工具
Search	搜索联机文档

(2) Build 工具栏与 Debug 工具栏。

Build 工具栏与 Debug 工具栏含有 8 个工具。当用户建立好应用程序项目后,可利用这些工具对项目进行编译、连接、调试和运行。常用工具按钮的功能如表 2-2 所示。

表 2-2 Build 工具栏与 Debug 工具栏中常用的工具按钮

工 具	功 能
Compile	编译文件, 要编译的文件必须是构成项目的一部分
Build	编译项目
Stop Build	停止编译项目
Start	执行程序
Break All	停止所有项目
Stop Debugging	停止调试
Restart	重新启动
Insert/Remove Breakpoint	插入或删除断点

3) 定制工具栏

工具栏也可以定制操作。下面以创建新的工具栏为例来介绍工具栏的定制过程，该工具栏名为“我的书签”，其中有相应的 5 个按钮用来管理、设置书签和书签的切换。其定制过程如下。

(1) 在工具栏选择快捷菜单中选择最下方的 Customize 命令，打开 Customize 对话框，如图 2-10 所示。单击 New 按钮打开 New Toolbar 对话框。

(2) 在 New Toolbar 对话框中的 Toolbar Name 文本框中输入“我的书签”，单击 OK 按钮。在 Visual Studio .NET 2008 开发环境中，从工具栏上任意选择几个需要的工具按钮拖放到“我的书签”工具栏中，这样就创建好了“我的书签”工具栏，如图 2-11 所示。

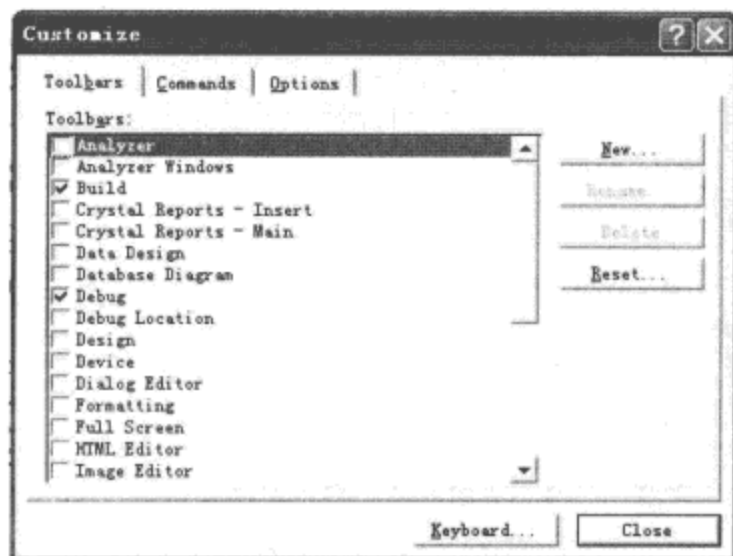


图 2-10 Customize 对话框

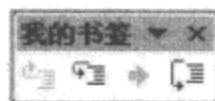


图 2-11 “我的书签”工具栏

在 Customize 对话框中选择需要删除的工具栏，并单击右侧的 Delete 按钮，就可以删除对应的工具栏。

2.1.7 解决方案资源管理器

项目是与应用程序相关的一组文件及其配置，用以生成最终的程序或二进制文件，一个应用程序对应一个项目。通过 Visual C++ .NET 2008 生成的应用程序其对应的项目文件则是通过解决方案来进行管理的，一个解决方案可以包含多个项目。解决方案是 Visual C++ .NET 2008 的一个重要组成部分，程序员的大部分工作都在 Visual C++ .NET 2008 中完成。Visual C++ .NET 2008 使用解决方案来组织项目、元素以及确定项目信息在屏幕上出现的方式。在一个解决方案中，可以处理每一个项目和项目对应的文件。

一个解决方案可包含由不同的开发工具包生成的项目，如 Visual C++ 和 Visual Basic 等。在桌面上，解决方案以窗口的方式组织项目、文件和项目设置。解决方案一般通过解决方

案管理器的形式显示于屏幕右侧，如图 2-12 所示。解决方案管理器窗口底部有一组标签，用于从不同的角度(视图)查看解决方案管理器的各个项目。解决方案管理器显示所创建的项目中包含的文件。展开文件树可以查看项目中所包含的各类文件，包括源(.cpp)文件、头(.h)文件和资源(.rc)文件。



图 2-12 解决方案管理器

解决方案文件以.sln 为后缀保存，项目文件以.prj 为后缀保存。要打开一个项目，只需打开相应的解决方案文件即可，如果直接打开项目文件，Visual C++ .NET 2008 会自动加载对应的解决方案。

解决方案管理器由视图窗口组成，每个视图窗口中都有一个相应的文件树，包含关于该项目的各种元素，展开文件树可以显示所选视图的详细信息。一个 Windows 应用程序的解决方案管理器一般包含图 2-12 所示的 3 种视图。

- **Class View(类视图):** 显示项目中定义的 C++ 类。展开文件树可以显示项目中定义的所有类，展开类可查看类的数据成员和成员函数以及全局变量、函数和类的定义。
- **Resource View(资源视图):** 显示项目中所包含的资源文件。展开文件树可显示所有类型的资源。
- **Solution Explorer(解决方案管理器):** 显示所创建的项目文件。展开文件树可以查看项目中所包含的文件。

2.1.8 类视图

类视图窗口将解决方案对应的项目以类的形式进行组织，展开类视图窗口中的树控件可以查看项目对应的所有类信息以及类对应的属性和方法，如图 2-13 所示。



图 2-13 类视图窗口

2.1.9 文件视图

Solution Explorer 视图显示解决方案、解决方案包含的项目文件和项目工作区中所包含的文件逻辑关系。一个解决方案可包含多个项目，其中有一个为活动项目，活动项目以高亮显示。

使用 Solution Explorer 视图可以完成查看文件、管理文件、增加文件、删除文件、移动文件、重命名文件和复制文件等工作。

要增加一个文件到项目中，可以通过右击解决方案并从快捷菜单中选择 **Add Existing Item** 子命令，从打开的文件对话框中选择相应的文件并添加；要从项目中删除一个文件，可右击需要删除的文件，然后从快捷菜单中选择 **Remove(移除)** 命令移除文件，或者通过选中要删除的文件并按 **Del** 键进行删除。



注意： 上述文件操作是相对项目而言的，例如，移除某文件，指的是从本项目中移除文件，而不是从磁盘上彻底删除该文件。因此还可以通过上述增加文件操作把该文件重新加入本项目。当然，如果需要从磁盘中彻底删除该文件，可从 Windows 系统的资源管理器中将其删除。

2.1.10 资源视图

资源视图显示解决方案对应的项目所包含的资源文件，展开文件树可显示所有相关的资源。

通过对图 2-3 所示的 New Project 对话框使用 Visual C++ .NET 2008 的向导可以自动为所生成的应用程序生成程序资源。默认生成的资源包括：图标、菜单、快捷键、工具栏和版本信息等。在 Visual Studio 主窗口左边选择 **Resource View** 选项卡切换到 Resource View

视图窗口, 可以查看生成资源的名称和类型。一般情况下, 都需对已生成的程序资源进行修改或添加新的资源。Resource View(资源视图)窗口如图 2-14 所示。

在资源窗口中, 我们可以创建和修改各种资源, 不同资源的创建、修改以及应用方式有所不同, 具体如下。

1. 创建新资源(插入资源)

在资源视图窗口中选择需要添加资源的项目, 然后在项目文件上右击, 从弹出的快捷菜单中选择 Add Resource 命令, 打开 Add Resource 对话框, 如图 2-15 所示。



图 2-14 Resource View 窗口

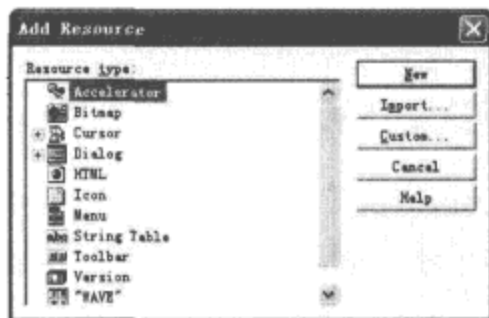


图 2-15 Add Resource 对话框

从 Resource type 列表中选择资源类型, 然后单击 New 按钮即可创建新的资源。

2. 查看和修改资源

打开 Resource View 窗口, 展开文件树, 从每一个分类中查看所有的资源。可以用菜单命令复制、移动、粘贴或删除资源, 也可以双击打开相应的编辑器来修改资源, 并运用资源属性对话框来修改资源的语言属性或条件属性。

3. 资源 ID

资源 ID 由映射到整数值上的字符串组成, 用于在源代码或资源编辑器中引用资源或对象。通过资源属性对话框, 可以改变资源的 ID 名或对应的整数值。具体的操作步骤如下。

- (1) 在 Resource View 窗口中选择要处理的资源。
- (2) 通过 View 菜单中的 Properties Windows(属性窗口)命令打开 Properties 窗口。
- (3) 在显示的 Properties 窗口中修改对应的 ID 子项就可以对资源 ID 进行修改, 如图 2-16 所示。

IDE 自动生成的符号名称前缀还用来代表资源或对象类型, 如光标前缀的 IDC_ ; 位图前缀的 IDB_ 等。可以利用 Visual Studio 提供的 Resource Symbols 对话框查看资源符号的定

义,还可以在 其中创建和删除资源符号,或者快速切换到某个资源对应的资源编辑器中。



图 2-16 用于修改资源 ID 的属性窗口

4. 对话框编辑器

在 Resource View 窗口中右击 Dialog(对话框)文件夹,在弹出的快捷菜单中选择 Add Dialog(添加对话框)命令打开对话框编辑器。

对话框编辑器用于创建或编辑对话框资源或对话框模板,向对话框中添加控件或调整其布局,并测试对话框的功能。另外,也可添加 ActiveX 控件,引入 Visual Basic 表单作为对话框资源,或者把对话框保存为模板以供将来使用。

图 2-17 显示的是对话框编辑器窗口,一旦打开此窗口,则对应的 Toolbox 工具箱中的 Dialog Editor 工具栏会同时处于可用状态,如图 2-18 所示。

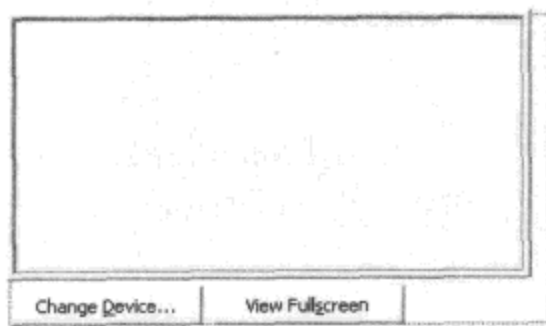


图 2-17 对话框编辑器窗口

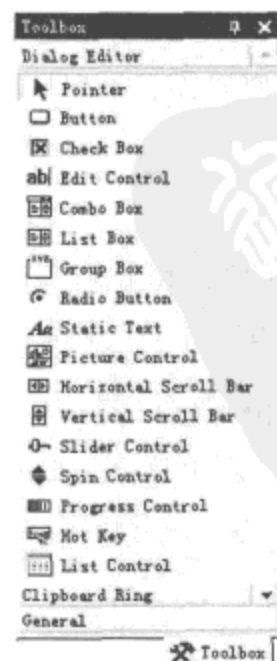


图 2-18 Toolbox 工具箱中的 Dialog Editor 工具栏

利用 Toolbox 工具箱的 Dialog Editor 选项卡可以向对话框添加所选择的控件。可以添加的控件有：静态文本(Static Text)、编辑框(Edit Control)、组合框(Group Box)、按钮(Button)、复选框(Check Box)、水平滚动条(Horizontal Scroll Bar)、垂直滚动条(Vertical Scroll Bar)、微调控件(Spin Control)、进展条(Progress Control)、热键(Hot key)、列表控件(List Control)、树形控件(Tree Control)、制表控件(Tab Control)和动画(Animation Control)等，还可以根据需要创建新的定制控件(Custom Control)。添加控件有以下两种方法。

- 将控件从 Toolbox 工具箱的 Dialog Editor 工具栏中拖到对话框编辑器窗口的指定位置。
- 先单击 Toolbox 工具箱的 Dialog Editor 工具栏中要添加的控件，然后移动光标到要添加控件的位置并单击鼠标。

要添加多个同一类型的控件，应同时按住 Ctrl 键。选择控件后，可以对其进行移动、复制、删除或调动，也可以拖动尺寸句柄缩放控件。另外，可以使用控件的属性窗口修改控件的属性。

通过 Layout 工具栏可以将对话框上对应的控件位置进行自动调整。该工具栏包括以下几个工具，如表 2-3 所示。

表 2-3 Layout 工具栏

工 具	功 能
Test Dialog	运行对话框，以测试对话框的外观和行为
Align Lefts	将选定的控件按左对齐格式放置
Align Rights	将选定的控件按右对齐格式放置
Align Tops	将选定的控件按上对齐格式放置
Align Bottoms	将选定的控件按下对齐格式放置
Center Vertical	将选定的控件按中心垂直对齐格式放置
Center Horizontal	将选定的控件按中心水平对齐格式放置
Space Across	使选定的控件两两水平间隔相同
Space Row	使选定的控件两两垂直间隔相同
Make Same Width	使选定的控件有相同的宽度
Make Same Height	使选定的控件有相同的高度
Make Same Size	使选定的控件有相同的宽度和高度
Toggle Grid	在显示或隐藏网格间进行切换
Toggle Guides	在显示或隐藏辅助线间进行切换

5. 菜单编辑器

通过选择 Resource View 窗口中的 Menu 文件夹, 然后右击, 从显示的快捷菜单中选择 Insert Menu 命令可以打开菜单编辑器。使用菜单编辑器可创建标准菜单和菜单项, 为菜单或菜单项定义热键、快捷键和状态栏提示, 或者创建快捷菜单。图 2-19 为打开某一菜单资源后的菜单编辑器。



图 2-19 菜单编辑器

1) 创建菜单项

创建菜单栏中菜单项的方法如下。

(1) 选择菜单编辑器中新建菜单项的方框, 或按 Tab 键(向左移)、Shift+Tab 键(向右移)或左右箭头键移到新建菜单项的方框(插入时按 Insert 键)。

(2) 使用鼠标双击需要设置新菜单的菜单位置, 菜单编辑器上显示可以输入状态, 输入菜单名就可以完成菜单生成。

(3) 打开属性窗口, 在 ID 子项中输入菜单项的 ID 号或选取已有的 ID 号。菜单首次创建时, IDE 可以根据选项名称自动生成一个 ID 值。

(4) 通过更改属性框中对应的属性来为菜单项指定风格。

2) 定义快捷键

利用菜单编辑器还可以定义快捷键, 具体操作步骤如下。

(1) 打开属性窗口, 选择需要设置快捷键的菜单项。

(2) 在 Caption 子项中将快捷键添加到菜单标题的后面。如果在菜单标题后输入转义符“\L”, 则所有快捷键都会按左对齐方式显示。

(3) 在快捷键编辑器中建立相应的快捷键表条目, 并赋予与菜单选项相同的 ID 号。

3) 创建快捷菜单

在应用程序中要使用快捷菜单, 首先要创建它, 然后再将其与应用程序代码链接。具体方法如下。

(1) 创建带空标题的菜单栏。

(2) 输入临时字符为菜单标题, 以便在菜单栏中创建菜单项。

- (3) 在菜单下创建快捷菜单的菜单选项。
- (4) 保存菜单资源。
- (5) 根据创建应用程序的不同,使用不同的方法将快捷菜单添加到程序中去,具体将在后文详细讲述。

6. 快捷键编辑器

定义了快捷键可以使用键盘上的按键来执行菜单或工具栏上的命令。使用快捷键编辑器可以添加、删除、更改和浏览项目所用到的快捷键,也可以查看和更改与快捷键表中每个条目有关的资源标识符,还可以为某个菜单选项定义快捷键。

通过在 Resource View 窗口中的 Accelerator 文件夹上选择 Insert Accelerator Table 命令或者双击 Accelerator 文件夹中的 Accelerator table 子项,可以打开快捷键编辑器。

图 2-20 所示为打开某一快捷键表资源后的快捷键编辑器。按 Insert 键或在表尾的方框中输入快捷键名,将弹出 Accel Properties 对话框,从中可以为快捷键表添加新的快捷键,可以在该对话框的 Key 列表框中输入键名,在 ID 列表框中输入快捷键标识符。

ID	Modifier	Key	Type
IDC_VIEWFULLSCREEN	Alt	VK_RETURN	VIRTKEY

图 2-20 快捷键编辑器

通过在快捷键编辑器的最后一行双击,可以进行新快捷键的编辑。快捷键编辑器中第一列用于设置快捷键对应的资源 ID,也就是设置快捷键为哪一个按钮或者菜单的快捷键,Key 列用于设置快捷键对应的虚拟键值,Type 列用于设置 Key 列对应使用的是虚拟键值还是 ASCII 码值。

Key 列键值的设置必须满足以下几个条件。

- 0~255 间的整数,可以写成十进制、十六进制或八进制格式。
- 单个键盘字符,大写的 A~Z 或数字 0~9 既可解释成 ASCII 码值,也可看成虚拟键值,其他字符都看成 ASCII 字符。
- 由单个 A~Z 间的字母键所产生的组合键对应 ASCII 码值。
- 任何合法的虚拟键标识符,可以单击 Key 列右侧的向下箭头来选择标准的虚拟键标识符。

当输入 ASCII 码值时,不能通过符号 A 和控制键的组合来产生相应的虚拟键值。

7. 字符串表编辑器

字符串表是一种 Windows 资源,包含应用程序用到的字符串表的 ID 号、值和标题。

每个应用程序只能有一个字符串表。字符串表编辑器用于编辑应用程序的字符串表资源。使用字符串表编辑器,可以浏览字符串表、向字符串表添加新的字符串,从字符串表中删除某一字符串,将字符串从某一段移到另一段,将字符串从某一资源文件移到另一资源文件,修改字符串及其标识符等。图 2-21 所示为打开某一应用程序的字符串表资源后的字符串表编辑器。通过在 Add Resource 对话框中选择新建 String Table(字符串表)或者从 Resource View 窗口中双击 String Table 文件夹中的 String Table 子项,都可以打开字符串编辑器。

ID	Value	Caption
IDS_STRING202	202	

图 2-21 字符串表编辑器

8. 版本信息编辑器

版本信息主要由公司名称、产品标识、产品版本号、版权和商标注册号组成。版本信息编辑器是用于编辑和维护版本信息的工具。每个应用程序只有一个版本信息资源,其名称为 VS_VERSION_INFO。在应用程序中调用函数 GetFileVersionInfo 和 VerQueryValue 可以访问版本信息。图 2-22 是打开某一应用程序的版本信息资源后显示的版本信息编辑器。

通过在 Add Resource 对话框中选择新建 Version(版本信息)或者从 Resource View 窗口中双击 Version 文件夹中已经存在的 Version 子项,都可以打开版本信息编辑器。

Key	Value
FILEVERSION	1. 0. 0. 1
PRODUCTVERSION	1. 0. 0. 1
FILEFLAGSMASK	0x17L
FILEFLAGS	0x1L
FILEOS	VOS_WINDOWS32
FILETYPE	VFT_APP
FILESUBTYPE	VFT2_UNKNOWN
Block Header	中文(中国) (080404b0)
Comments	
CompanyName	
FileDescription	DirectX9 Application
FileVersion	1. 0. 0. 1
InternalName	DirectX9
LegalCopyright	Copyright (C) 2004
LegalTrademarks	
OriginalFilename	DirectX9.exe
PrivateBuild	
ProductName	DirectX9 Application
ProductVersion	1. 0. 0. 1
SpecialBuild	

图 2-22 版本信息编辑器

9. 图形编辑器

图形编辑器如图 2-23 所示,由一套功能强大的绘图工具组成,用于绘制位图、图标和光标。通过在 Add Resource 对话框中选择新建图形资源(Bmp 资源、Icon 资源等)或者从 Resource View 窗口中双击已经存在的图形资源子项,都可以打开图形编辑器。

图形编辑器打开后,将自动激活 Graphics 工具栏和 Colors 工具栏。Graphics 工具栏由

选择(Select)、自由选择(Select Region)、颜色拾取(Select Color)、橡皮(Eraser)、填充(Fill)、铅笔(Pencil)、画刷(Brush)、喷枪(Airbrush)、画线(Line)、文字(Text)、空心矩形(Rectangle)、实心矩形(Filled Rectangle)和空心椭圆(Ellipse)等 21 个工具组成, 供用户绘制图形, 其功能与一般绘图软件工具栏一样, 这里不再赘述。同样, Colors 工具栏供用户选择绘图时所用的颜色。

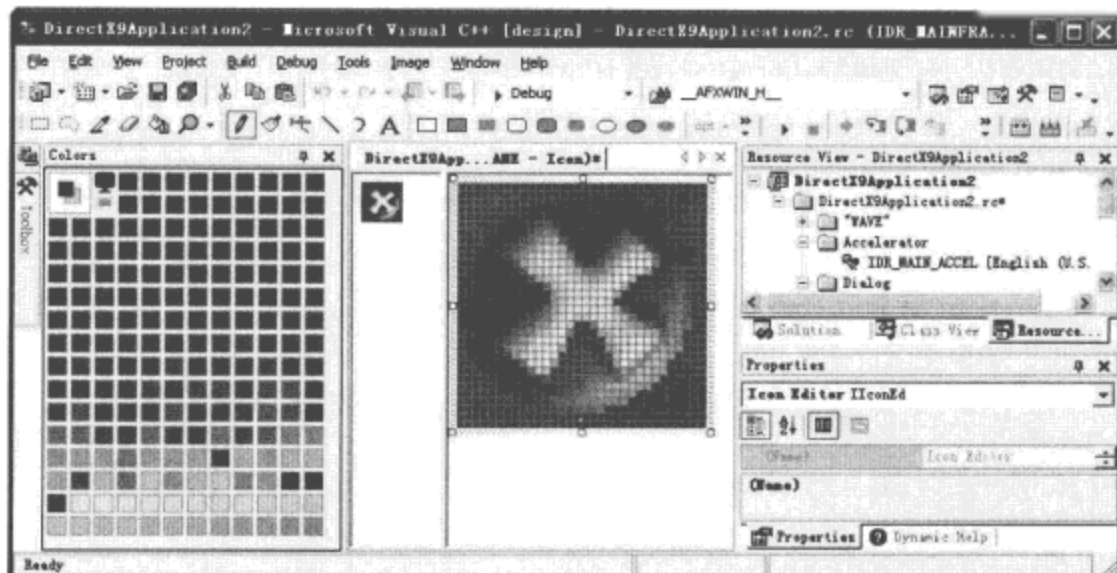


图 2-23 图形编辑器

10. 工具栏编辑器

工具栏编辑器用于创建工具栏资源, 并可以将已有位图转换为工具栏资源。工具栏编辑器以图形方式显示要处理的工具栏及已被选择的工具按钮图像。通过在 Add Resource 对话框中选择新建 Toolbar 资源或者从 Resource View 窗口中双击已经存在的 Toolbar 子项, 都可以打开工具栏编辑器。图 2-24 所示为打开某一工具按钮后的工具栏编辑器。

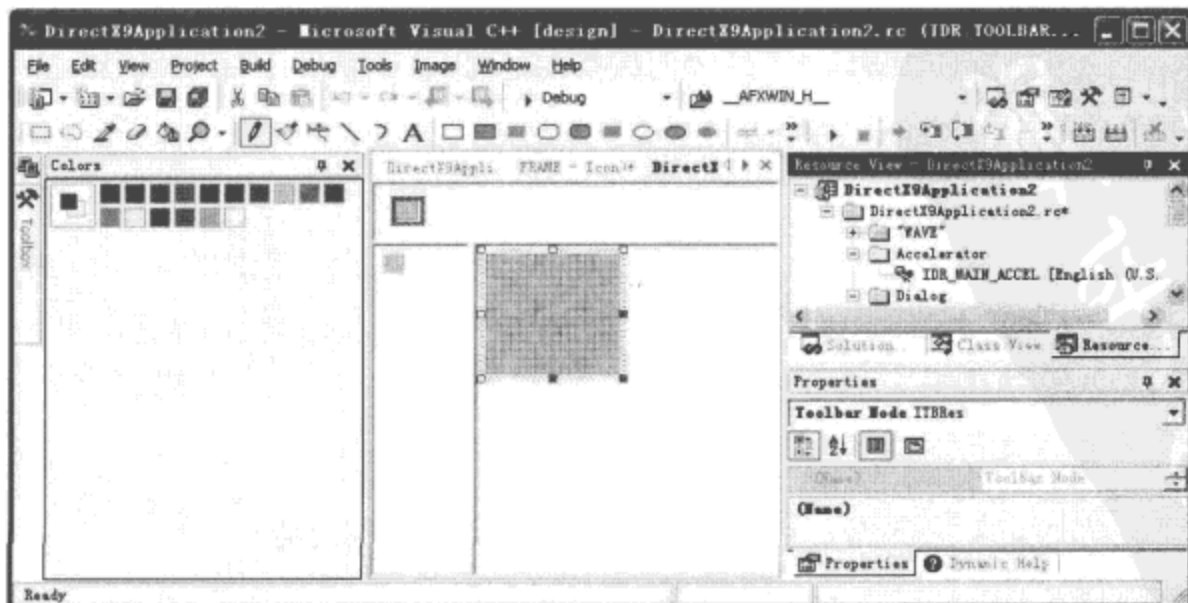


图 2-24 工具栏编辑器

2.1.11 帮助文档的使用

在实际开发工作中，任何一个程序设计工作者都离不开帮助文档，没有一个程序员可以把所有的函数都记在大脑中，因此一个好的在线帮助是必不可少的。Windows API 函数成百上千，MFC 类库也十分庞大，而且在不断地扩充，要求程序开发人员完全记住每个函数或类的细节是不可能的事情。在我们头脑中有一些印象后，应该充分利用联机帮助提供的方便，查询函数的参数及其用法和类的相关信息。在开发过程中，当遇到不熟悉的函数时，可以将光标放在对应的函数或变量类型描述符上，按下键盘上的 F1 键，这时，耐心等待片刻，关于该函数或类型的帮助信息就会在一个新的窗口中出现了(这一切的前提是，在用户安装开发工具的时候，同时也选择了安装帮助文档)。帮助文档的格式是网页模式的 CHM 文件，因此在内部可以进行各种链接查看，非常方便。

Visual C++ .NET 2008 联机帮助系统允许用以下 5 种方式获得帮助。

- 通过目录获得帮助。选择 Help 菜单的 Contents 命令，帮助窗口切换到 InfoView 模式，帮助文档是按照章节(目录)的层次结构组织的。
- 通过索引获得帮助。选择 Help 菜单的 Index 命令打开帮助窗口(若帮助窗口已打开则直接选择 Index 标签)，在文本框中输入索引的关键字，便可查到相关的主题或函数。
- 通过搜索获得帮助。选择 Help 菜单的 Search 命令打开帮助窗口(若帮助窗口已打开则直接选择 Query 标签)，在文本框中输入要搜索的单词，便可查到需要的主题或文本。
- 通过书签获得帮助。书签是一个标记，用户可以给任何一篇感兴趣的相关帮助信息的文本加上一个标签，下一次便可以通过书签快速地直接显示出该书签所指的文本。
- 按 F1 键获得帮助，在编程过程中，只要把光标移到需要查询的函数或类中，然后按下 F1 键，帮助系统便会启动。并直接跳至有关该函数或类的说明内容的文本中。
- 使用动态帮助。动态帮助是 Visual C++ .NET 2008 提供的新的 IDE 功能，通过在 Visual C++ .NET 2008 开发环境的文档中选择需要查询的函数，动态帮助系统将自动帮助用户定位到对应的函数说明文档。

2.2 Linux 下的开发环境

在各种版本的 Linux 中，Ubuntu 是一个使用方便的版本，因此就以 Ubuntu 为例进行开

发环境配置的相关说明。安装 Ubuntu 比较容易，安装的光盘可以在其官方网站直接下载。有两种格式的安装光盘，一种是 CD 的，一种是 DVD 的，很明显 DVD 格式的光盘包含的软件多，不过即使只有 CD 格式的安装盘也不用担心，以后很多软件都可以在网络上直接安装。具体的安装步骤如下。

(1) 将 Ubuntu 的安装盘放在光盘驱动器中，启动电脑，并选择由光盘启动计算机，开始画面是英文的，按键盘上的 F2 键后出现语言选择菜单，选择简体中文。

(2) 选择将 Linux 安装到硬盘。

(3) 如果在计算机中，同时还安装了其他的操作系统，则建议先仔细查阅资料，了解如何进行磁盘分区。若该计算机仅安装了 Linux 系统，则直接选择使用整个硬盘(建议初学者在 Windows 系统下安装虚拟机，再在虚拟机上安装 Linux)。

(4) 在提示创建用户的时候，要注意记录安装过程中创建用户名和密码(重要提示：即使用户写的用户名是大写的，系统也默认改成小写的)。

(5) 其他一直单击“下一步”按钮即可。

(6) 系统安装完成后，还不能开发 C/C++ 程序，需要进行后续安装。利用 Vi 编辑 `/etc/apt/source.list` 文件，可先将所有的行全都加注释，然后保存文件，之后执行 `sudo apt-cdrom add` 指令，指定安装软件由光盘提供。

(7) 安装开发系统，执行 `sudo apt-get install buildXXX` 指令，安装开发工具，在安装完成后，执行 `g++` 指令，如提示没有输入文件，则证明安装完毕；如提示错误的指令，则说明开发系统未安装好，检查 6~7 步骤，重新操作，直到安装成功。

2.2.1 Vi 编辑器的基本使用

系统安装完毕后，在用户目录(`/home/yourusername/`)下创建一个开发用的目录(`mkdir XXXX`)，进入目录(`cd XXXX`)然后利用 Vi 编辑器创建应用程序，命令是 `vi XXX.CPP`。

在 Vi 系统中，有两个工作状态，默认是 Vi 的命令状态(注意，不是系统的 shell 命令状态)，在命令状态下，我们能做类似于文件保存等动作，但不能编辑文件。我们按键盘上的 I 键或者 Insert 键，进入编辑状态，此时可以编辑源代码程序。在编辑状态下，我们随时可以按 Esc 键，切换回命令状态。

2.2.2 Vi 编辑器的命令

Vi 编辑器的命令很多，利用这些命令，可以方便地进行各种文本编辑，虽然 Vi 没有可视化的界面，但是，一个熟练应用 Vi 的用户，编辑代码的效率一点也不比利用可视化工具的用户速度慢，相反会快一些。以下是常用的 Vi 命令。

1. 进入 Vi 的命令

- `vi filename`: 打开或新建文件, 并将光标置于第一行首。
- `vi +n filename`: 打开文件, 并将光标置于第 `n` 行首。
- `vi + filename`: 打开文件, 并将光标置于最后一行首。
- `vi +/pattern filename`: 打开文件, 并将光标置于第一个与 `pattern` 匹配的串处。
- `vi -r filename`: 在上次正用 Vi 编辑时发生错误, 恢复 `filename`。
- `vi filename...filename`: 打开多个文件, 依次进行编辑。

2. 移动光标类命令

- `h` 或 `Backspace`: 光标左移一个字符。
- `l` 或 `Space`: 光标右移一个字符。
- `k` 或 `Ctrl+p`: 光标上移一行。
- `j` 或 `Ctrl+n` 或 `Enter`: 光标下移一行。
- `w` 或 `W`: 光标右移一个字至字首。
- `b` 或 `B`: 光标左移一个字至字首。
- `e` 或 `E`: 光标右移一个字至字尾。
- `)`: 光标移至句尾。
- `(`: 光标移至句首。
- `}`: 光标移至段落开头。
- `{`: 光标移至段落结尾。
- `nG`: 光标移至第 `n` 行首。
- `n+`: 光标下移 `n` 行。
- `n-`: 光标上移 `n` 行。
- `n$`: 光标移至第 `n` 行尾。
- `H`: 光标移至屏幕顶行。
- `M`: 光标移至屏幕中间行。
- `L`: 光标移至屏幕最后行。
- `0`(注意是数字零): 光标移至当前行首。
- `$`: 光标移至当前行尾。

3. 屏幕翻滚类命令

- `Ctrl+u`: 向文件首翻半屏。
- `Ctrl+d`: 向文件尾翻半屏。
- `Ctrl+f`: 向文件尾翻一屏。



- Ctrl+b: 向文件首翻一屏。
- nz: 将第 n 行滚至屏幕顶部, 不指定 n 时将当前行滚至屏幕顶部。

4. 插入文本类命令

- i: 在光标前。
- I: 在当前行首。
- a: 在光标后。
- A: 在当前行尾。
- o: 在当前行之下新开一行。
- O: 在当前行之上新开一行。
- r: 替换当前字符。
- R: 替换当前字符及其后的字符, 直至按 Esc 键。
- s: 从当前光标位置处开始, 以输入的文本替代指定数目的字符。
- S: 删除指定数目的行, 并以所输入的文本代替。
- ncw 或 nCW: 修改指定数目的字。
- nCC: 修改指定数目的行。

5. 删除命令

- ndw 或 ndW: 删除光标处的字符及其后的 n-1 个字。
- do: 删至行首。
- d\$: 删至行尾。
- ndd: 删除当前行及其后 n-1 行。
- x 或 X: 删除一个字符, x 删除光标后的, 而 X 删除光标前的。
- Ctrl+u: 删除输入方式下所输入的文本。

6. 搜索及替换命令

- /pattern: 从光标开始处向文件尾搜索 pattern。
- ?pattern: 从光标开始处向文件首搜索 pattern。
- n: 在同一方向重复上一次搜索命令。
- N: 在反方向上重复上一次搜索命令。
- :s/p1/p2/g: 将当前行中所有的 p1 均用 p2 替代。
- :n1,n2s/p1/p2/g: 将第 n1 至 n2 行中所有的 p1 均用 p2 替代。
- :g/p1/s//p2/g: 将文件中所有的 p1 均用 p2 替换。

7. 选项设置

- all: 列出所有选项设置情况。

- **term**: 设置终端类型。
- **ignorance**: 在搜索中忽略大小写。
- **list**: 显示制表位(Ctrl+I)和行尾标志(\$)。
- **number**: 显示行号。
- **report**: 显示由面向行的命令修改过的数目。
- **terse**: 显示简短的警告信息。
- **warn**: 在转到别的文件时若没保存当前文件则显示 No write 信息。
- **nomagic**: 允许在搜索模式中, 使用前面不带“\”的特殊字符。
- **nowrapscan**: 禁止 Vi 在搜索到达文件两端时, 又从另一端开始。
- **mesg**: 允许 Vi 显示其他用户用 write 写到自己终端上的信息。

8. 最后行方式命令

- **:n1,n2 co n3**: 将 n1 行到 n2 行之间的内容拷贝到第 n3 行下。
- **:n1,n2 m n3**: 将 n1 行到 n2 行之间的内容移至第 n3 行下。
- **:n1,n2 d**: 将 n1 行到 n2 行之间的内容删除。
- **:w**: 保存当前文件。
- **:e filename**: 打开文件 filename 进行编辑。
- **:x**: 保存当前文件并退出。
- **:q**: 退出 Vi。
- **:q!**: 不保存文件并退出 Vi。
- **!:command**: 执行 shell 命令 command。
- **:n1,n2 w!command**: 将文件中 n1 行至 n2 行的内容作为 command 的输入并执行, 若不指定 n1 和 n2, 则表示将整个文件内容作为 command 的输入。
- **:r!command**: 将命令 command 的输出结果放到当前行。

2.2.3 Vi 编辑器环境设置

通常情况下可以用 vim(一个 Vi 的升级版)进行开发。

vim 初始配置文件位置“`~/.vimrc`”一般 vim 会有一个默认的配置文件的样本。用户在使用时一般会 cp 到用户目录中, 然后再修改(`"cp /usr/share/vim/vim70/vimrc_example.vim ~/.vimrc"`), 这个文件通常就能达到用户的基本要求, 不需要做太多的修改(如果设置完后, 发现功能没有起作用, 请检查系统是否安装了 vim-enhanced 包)。

配置文件内容如下。

```
set nocompatible "去掉有关 vim 一致性模式, 避免以前版本的一些 bug 和局限
set number "显示行号
```



```

set background=dark "背景颜色暗色(可以保护眼睛, 因此建议这样设置)
syntax on "语法高亮显示(这个一般都是要的)
set history=50 "设置命令历史记录为 50 条
set autoindent "使用自动对齐, 也就是把当前行的对齐格式应用到下一行
set smartindent "依据上面的对齐格式, 智能地选择对齐方式
set tabstop=4 "设置 Tab 键为 4 个空格
set shiftwidth=4 "设置当行之间交错时使用 4 个空格
set showmatch "设置匹配模式, 例如输入一个左括号时会匹配右括号
set incsearch "搜索选项(比如, 输入"/bo", 光标会自动找到第一个"bo"所在的位置)
"是否生成一个备份文件(备份的文件名为原文件名加 "~" 后缀)
"(不喜欢这个备份设置的可以注释掉)

"if has("vms")
" set nobackup
"else
" set backup
"endif

```

2.2.4 g++编译程序的方法

编辑第一个 C 程序时, 可以只做一个最简单的输出程序, 然后将 Vi 程序切换到命令状态, 输入:wq 命令保存程序并退出编辑器, 然后再使用命令 g++ XXXX.cpp 编译程序。如果程序没有错误则系统不做任何提示, 并可看到当前目录多了一个 a.out 文件, 运行该程序 (./a.out) 可查看执行结果。如果编译的文件是纯 C 语言写的, 也可以使用 gcc(而不用 g++) 进行编译。

2.2.5 g++编译程序的选项

开发工作者在实际开发的时候, 由于使用了不同的工具库或不同的开发模式, 因此在编译时的选项也有所不同。下面是常用的编译选项, 用户可以根据需要选择不同的项目。

1) -x language filename

设定文件所使用的语言, 使后缀名无效, 对以后的多个有效。也就是根据约定, C 语言的后缀名称是.c, 而 C++的后缀名是.C 或者.cpp, 如果一定要指定的 C 代码文件的后缀名是.pig, 编译程序也不认为有什么不妥, 只要用户喜欢就可以。此时就可以用这个参数, 这个参数对它后面的文件名都起作用, 除非到了下一个参数的使用。可以使用的参数还有更多。

例如:

```
gcc -x c hello.pig
```

2) -x none filename

关掉上一个选项，也就是让 gcc 根据文件名后缀，自动识别文件类型。

例如：

```
gcc -x c hello.pig -x none hello2.c
```

3) -c

只激活预处理，编译和汇编，也就是它只把程序做成 obj 文件。

例如：

```
gcc -c hello.c //将生成.o 的 obj 文件
```

4) -S

只激活预处理和编译，就是它只把文件编译成为汇编代码。

例如：

```
gcc -S hello.c //将生成.s 的汇编代码，你可以用文本编辑器察看
```

5) -E

只激活预处理，这个不生成文件，你需要把它重新定向到一个输出文件里面。

例如：

```
gcc -E hello.c > pianoapan.txt  
gcc -E hello.c | more
```

(只要有耐心，用户可以看到，一个 hello word 也要预处理成 800 行的代码。)

6) -o

制定目标名称，缺省的时候 gcc 编译出来的文件是 a.out。

例如：

```
gcc -o hello.exe hello.c (例如一个 windows 用习惯了的用户，可以这样设置)  
gcc -o hello.asm -S hello.c
```

7) -pipe

使用管道代替编译中的临时文件，在使用非 GNU 汇编工具的时候，可能有些问题。

例如：

```
gcc -pipe -o hello.exe hello.c
```

8) -ansi

关闭 GNU C 中与 ANSI C 不兼容的特性, 激活 ANSI C 的专有特性(包括禁止一些 `asm` `inline` `typeof` 关键字以及 `Unix`、`vax` 等预处理宏)。

9) -fno-asm

此选项实现 `ansi` 选项功能的一部分, 它禁止将 `asm`, `inline` 和 `typeof` 用作关键字。

10) -fthis-is-variable

就是向传统 C++ 看齐, 可以使用 `this` 当一般变量使用。

11) -fcond-mismatch

允许条件表达式的第二和第三参数类型不匹配, 表达式的值将为 `void` 类型。

12) -funsigned-char -fno-signed-char -fsigned-char -fno-unsigned-char

这四个参数是对 `char` 类型进行设置, 决定将 `char` 类型设置成 `unsigned char`(前两个参数)或者 `signed char`(后两个参数)类型。

13) -include file

包含某个代码, 简单来说, 当某个文件需要另一个文件的时候, 就可以用它设定, 功能就相当于在代码中使用 `#include <filename>` 语句。

例如:

```
gcc hello.c -include /root/pianopan.h
```

14) -imacros file

将 `file` 文件的宏扩展到 `gcc/g++` 的输入文件, 宏定义本身并不出现在输入文件中。

15) -Dmacro

相当于 C 语言中的 `#define macro` 语句。

16) -Dmacro=defn

相当于 C 语言中的 `#define macro=defn` 语句。

17) -Umacro

相当于 C 语言中的 `#undef macro` 语句。

18) **-undef**

取消对任何非标准宏的定义。

19) **-Idir**

在用户使用 `#include "file"` 语句的时候, gcc/g++ 会先在当前目录查找用户所制定的头文件, 如果没有找到, 编译系统会回到默认的头文件目录中找。如果使用 **-I** 制定了目录, 则编译系统会先在用户所指定的目录中查找, 然后再按常规的顺序去找。

若用户使用 `#include <file>` 语句, gcc/g++ 会到 **-I** 指定的目录查找, 若查找不到则到系统默认的头文件目录中查找。

20) **-I-**

取消前一个参数的功能, 所以一般在 **-Idir** 之后使用。

`-idirafter dir`

在 **-I** 的目录里面查找失败, 将到这个目录里面查找。

`-iwithprefix dir`

一般一起使用, 当 **-I** 的目录查找失败时, 会到 `prefix+dir` 下查找。

21) **-nostdinc**

使编译器不在系统默认的头文件目录里面找头文件, 一般和 **-I** 联合使用, 明确限定头文件的位置。

22) **-nostdin C++**

规定不在 g++ 指定的标准路径中搜索, 但仍在其他路径中搜索。此选项在使用 `libg++` 库时使用。

23) **-C**

在预处理的时候, 不删除注释信息, 一般和 **-E** 使用, 有时候分析程序。

24) **-M**

生成文件关联的信息。包含目标文件所依赖的所有源代码例如: `gcc -M hello.c`。

25) **-MM**

和上面 **-M** 一样, 但是它将忽略由 `#include <file>` 造成的依赖关系。

26) -MD

和-M 相同，但是输出将导入到.d 的文件里面。

27) -MMD

和-MM 相同，但是输出将导入到.d 的文件里面。

28)-Wa, option

此选项传递 option 给汇编程序；如果 option 中间有逗号，就将 option 分成多个选项，然后传递给汇编程序。

29) -Wl.option

此选项传递 option 给连接程序；如果 option 中间有逗号，就将 option 分成多个选项，然后传递给连接程序。

30) -llibrary

指定编译的时候使用的库。

例如：

```
gcc -lcurses hello.c      //使用 ncurses 库编译程序
```

31) -Ldir

指定编译的时候搜索库的路径。比如用户自定义的库，可以用它指定目录，不然编译器将只在标准库的目录中找。dir 就是目录的名称。

32) -O0 -O1 -O2 -O3

编译器优化选项的 4 个级别，-O0 表示不优化，-O1 为默认值，-O3 的级别最高。

33) -g

只是编译器在编译的时候产生调试信息。

34) -gstabs

此选项以 stabs 格式声称调试信息，但是不包括 gdb 调试信息。

35) -gstabs+

此选项以 stabs 格式声称调试信息，并且包含仅供 gdb 使用的额外调试信息。

36) -ggdb

此选项将尽可能地生成 gdb 可以使用的调试信息。

37) -static

此选项将禁止使用动态库，所以编译出来的东西一般都很大，也不需要什么动态连接库就可以运行。

38) -share

此选项将尽量使用动态库，所以生成文件比较小，但是需要系统有动态库。

39) -traditional

试图让编译器支持传统的 C 语言特性。

2.2.6 运行应用程序

运行应用程序非常简单，直接指定路径及文件名，即可运行程序。例如若应用程序在 /home/user/app 路径下，程序名为 test.o，只要输入下列命令即可运行程序。

```
/home/user/app/test.o
```

2.2.7 帮助文档的使用

Linux 环境下帮助文档的使用也很简单，只要在命令行状态下输入下列指令即可启动相应的帮助。

```
man XXX
```

命令中的 XXX 即为需要查询的函数。有关帮助文档的更多使用技巧，请参考 Linux 手册。

2.3 CodeBlocks 集成开发工具介绍

首先安装 Code::Blocks，这个过程比较简单，这里就不多叙述，按以下步骤操作即可。

- 安装 Code::Blocks 第一个正式版本 Ver 8.02。
- 安装 Code::Blocks 的最新升级包。
- 安装 Code::Blocks 的简体中文语言包。

- 完成 Code::Blocks 的基本配置。

2.3.1 创建工程

双击桌面上的 CodeBlocks 图标，会出现 CodeBlocks 的启动界面，如图 2-25 所示。

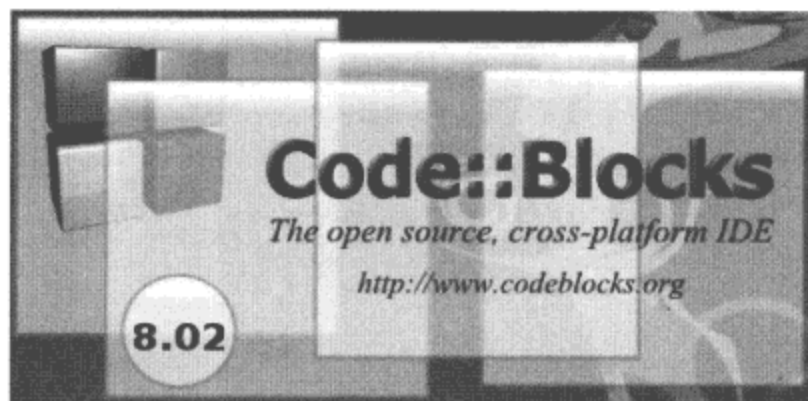


图 2-25 CodeBlocks 的启动界面

进入系统后，选择“文件”→“新建”→“创建新项目”命令，然后根据需要选择项目种类，即可完成工程的创建，这个过程与 Visual Studio .NET 的使用非常类似。

2.3.2 创建文件

创建工程后，就可为工程添加文件，进行开发工作。这里要注意，在 Linux 系统下，如果不创建项目就直接创建文件，那么在开发过程中，会遇到一移动鼠标 CodeBlocks 就关闭的一个错误。因此千万要记住，先创建项目，再创建文件。

一般的，创建 CPP 文件进行代码编写工作，所有的程序设计方法完全按照 C++ 的规范进行就可以了。代码编写完毕后选择保存，即可存储源文件或其他文件。

2.3.3 项目属性设置

与 Visual Studio .NET 类似，项目的属性也会根据实际情况改变，CodeBlocks 一样给用户提供了设置属性的界面，如图 2-26 所示。

在当前对话框中，单击“项目的构建选项”按钮，会出现“项目 build 选项”对话框，如图 2-27 所示。

这里的设置功能与 Visual Studio .NET 的作用完全相同，设置注意事项也类似，因此就不再赘述。

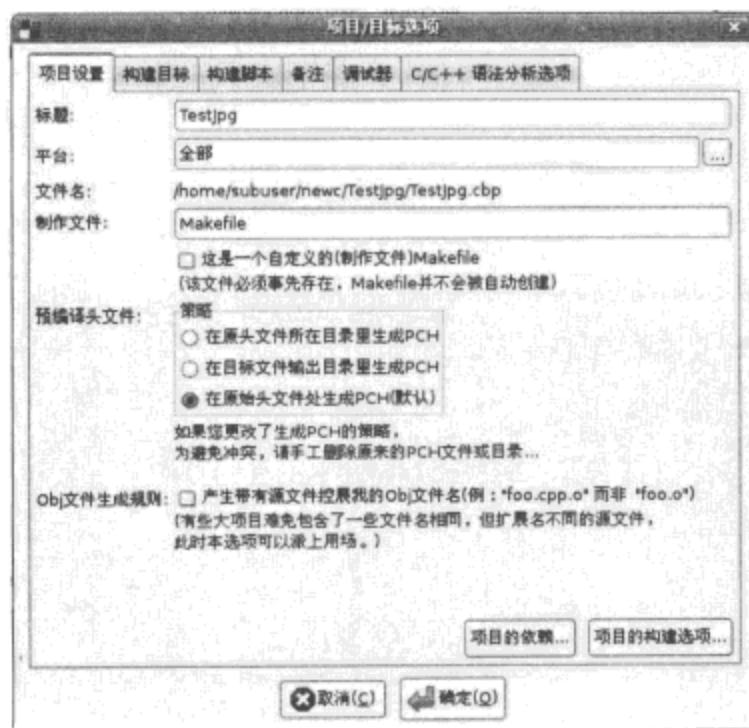


图 2-26 项目属性设置

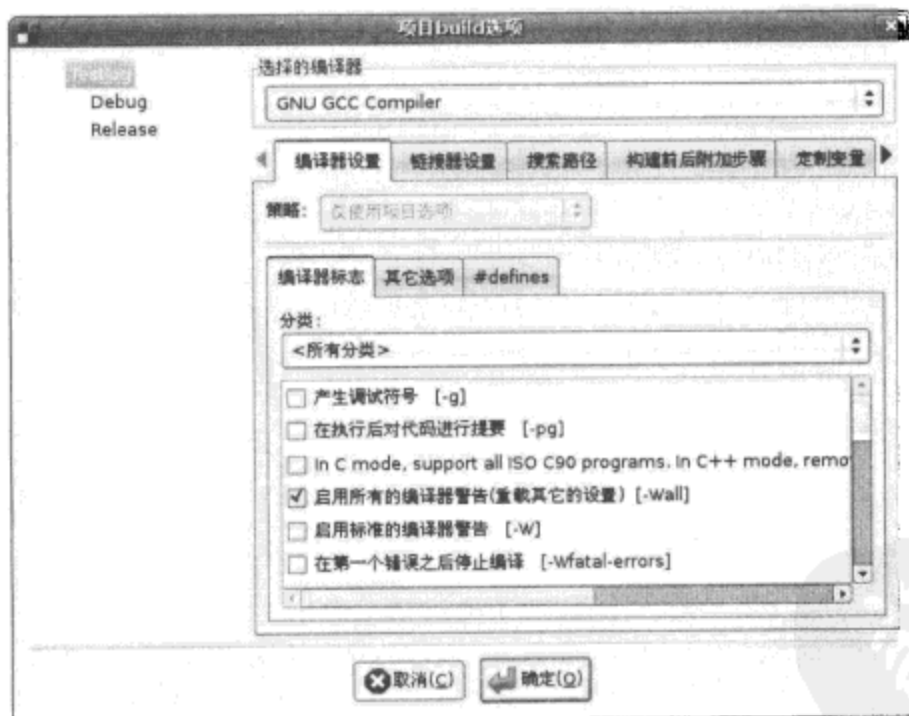


图 2-27 “项目 build 选项”对话框

2.3.4 编译及运行

在 CodeBlocks 下编写完项目后，选择“构建”→“构建”命令即可编译当前工程。编译成功后，选择“构建”→“运行”命令即可运行程序。这个步骤也与 Visual Studio .NET 相当接近，读者可自行实验其他功能。

2.4 绘图函数库的使用

现在的计算机都支持图形显示,但由于图形的绘制在不同的系统中方式不一样,因此标准的 C/C++ 中没有与绘图相关的函数。本书有很多例程序,都用图形的方式演示程序的执行过程或结果,因此为了显示这些图形,必须提供一种方便的图形显示方法。在不同的系统下,有不同的窗口显示 API 或图形 API,而 C++ 程序可以在常用的各种操作系统或计算机平台上工作。为了保证各种平台下工作的用户都能得到正确的运行结果,本书使用 OpenGL 这种跨平台的图形 API。下面简单介绍一下 OpenGL 开发库的安装。

首先安装 OpenGL 基本开发库,只要将 gl 与 glu 的函数声明文件与库文件放在一个自建的目录下即可。将 GLUT 库对应的文件也复制到基本开发库对应的目录下,然后在开发环境中配置附加包含目录与附加库目录,在程序中直接引用函数声明文件,并调用相关函数即可使用。

关于如何使用 OpenGL 开发库进行绘图,将在第 9 章讲述动态分配内存的应用时,进行详细的说明。

本章小结

Visual C++ .NET 2008 集成开发环境为程序开发人员提供了一个功能强大的可视化开发工具。虽然在 IDE 环境里能完成的所有功能,都可以通过命令行来实现,但使用 IDE 集成的各个工具,能够快速地完成具体的项目,还能根据要求定制适合的编程环境,实现个性化的定制(定制菜单和定制工具栏)。

在此环境中,用户通过菜单或工具栏实现各种命令,完成项目的创建、设置与管理、编译与调试。特别是使用 IDE 提供的各种类型的应用程序向导,能让开发人员方便、快速地生成完整的各种类型程序框架,这就使得开发人员能够把精力集中于项目的实现过程。

使用 IDE 的资源编辑器能方便地创建、修改、编辑和管理程序中所使用的各种资源,如快捷键、对话框、图标、菜单、字符串和工具栏等。熟练地掌握 IDE 提供的各种工具,有助于提高编程效率。

使用 Linux 环境进行程序开发的用户,可以使用 Vi 编辑器及 g++ 编译程序来编辑和编译 C++ 程序,一样可以进行开发工作;而习惯于可视化工具开发,又想在 Linux 下工作的用户,可以选择 CodeBlock 作为开发工具,这个集成环境的应用,基本和 Visual C++ .NET 2008 是一样的。

第 3 章 基本数据类型

本章内容:

- C++支持的字符集。
- 基本数据类型。
- 标准输入和输出。

重点:

各数据类型的使用方法和内存结构

目的:

掌握最基本的 C++类型和基本程序语句, 完成一个最简单的 C++程序。

学习任何一种语言, 首先都要掌握该语言最基本的语言要素, 而字符集、关键字和基本数据类型, 正是 C/C++语言的基本语言要素。在本章中, 将为大家介绍 C++的字符集、关键字和基本数据类型。

3.1 基本程序组成结构

3.1.1 一个基本的 C++程序

首先来看一个最简单的 C++程序, 通过这个简单的程序, 可以看到一个 C++程序的最基本组成。

```
#include <stream.h>
int main()
{
    int a;
    float b;
    int d;
    const int c = 4;
    cin>>a>>b;
```



```

    d = a+3;
    cout<<"a = "<<a<<"b = "<<b<<"c = "<<c<<"d = "<<d<<endl;
    return 0;
}

```

程序结构解析如下。

第一行的语句，将在编译预处理中进行介绍，这里先忽略。

第二行是程序的开始，所有的程序都由 `main` 语句开始^①。

第三行到第五行，定义了三个变量：`a`、`b`、和 `d`。

第六行定义了一个常量 `c`。

第七行执行了数据的输入操作。

第八行执行了一些运算操作。

第九行是将所有的数据进行输出(即在屏幕上进行显示)。

第十行表示程序结束。

以上是一个最简单的 C++ 程序的组成结构，所有的程序，不管多复杂，都是由这些基本的元素组成。程序中每条语句的具体执行原理，以及有关常量和变量、运算表达式、输入输出等细节，将在本章后续小节中进行详细描述。下面对每个组成结构的细节进行详细的说明。

3.1.2 基本输入输出

程序的运行结果最终要输出到某种介质载体上以备人们使用，本节将向读者介绍在控制台程序中通过屏幕进行输入和输出与用户进行交互的方法。

1. I/O 流输入输出

在 C 语言中，使用的是被称为格式化输入输出函数的标准库函数来完成大部分 I/O 操作的；而在 C++ 中，引入了输入输出流的概念，在后面第 17 章将专门来讨论输入输出流。现在先简单了解如何使用 C++ 的输入输出，为后面学习其他的内容打好基础。

首先看一个向屏幕进行输出的例子，通过这个例子来详细解释 C++ 中的输出操作。

```

#include <iostream>                                // 本行代码用来包含 C++ 标准的库函数和类定义

```

① 并不是所有的入口函数都是 `int main()` 这样的形式，事实上，标准 C/C++ 还支持带参数的 `main`，其形式是：`int main(int argc, char **argv)`，这样的函数在第 7 章中就会用到，请读者注意。

另：`main` 函数的返回值，按照 C/C++ 的初衷，应该是 `int` 类型，但在 Windows 系统下，函数声明为 `void main()` 形式，也是被允许的。关于 `void` 的更多说明，请参考第 8 章相关内容。

```

using namespace std;           // 指定所使用的名空间(当使用包含的是
                                // <iostream>而不是传统的<iostream.h>文件时需要指
                                // 定名空间), 有关头文件包含和名空间的具体内容参
                                // 见后面章节
void main()                    // 程序主函数定义, 每一个程序都要有一个主函数,
                                // 这是程序的入口点, 程序从主函数这里开始执行
{
    int idata;
    char ch='a';
    float fdata=34.343423433222;
    cout << "This is an I/O test!" << endl;
    cout<<"18"<<endl;
    idata = 18;
    cout<<"The number of idata is "<<idata<<endl;
    cout<<ch<<endl;
    cout<<"The number of fdata is "<< fdata <<endl;
}

```

在这个程序中定义了三个变量, 分别是整型变量 `idata`、字符型变量 `ch` 和实型变量 `fdata`, 然后通过语句:

```
cout<<"This is an I/O test!" << endl;
```

向屏幕输出:

```
This is an I/O test!
```

在这句代码中, “`cout`”是标准输出流对象, 也就是在语句中代表标准的输出设备(一般是显示器屏幕); “`<<`”为流插入操作符, 表示将它后面跟着的数据输出至前面的输出对象中, 这里用来向屏幕进行显示的操作。流插入操作符也可以用在不同的输出流对象上(如文件、打印机等), 而不仅仅是标准输出, 且输出可以级联操作。

所以, 在一个“`<<`”流插入操作符和数据后面还可以继续使用流插入操作符接着输出。这里接下来输出的 `endl` 代表 `end of line`, 即本行结束换下一行输出的意思。如: `cout<<"18"<<endl;` 即输出字符“18”后换行。下一行的 `cout<<"The number of idata is" <<idata<<endl;` 输出结果为 `The number of idata is 18`。

要注意的是, 本行的 18 和上面输出的 18 是不一样的, 前面输出的 18 是一个字符串, 本行输出的是变量 `idata` 的值 18。

这表明, 在输出的时候不必再指明输出数据的数据类型, 对于基本数据类型, `cout` 能够自动识别其类型并进行相应的输出处理。但是, 当需要输出自定义的数据类型(如自定义结构、类)时, `cout` 是无法自动完成这件事的, 必须要对 `cout` 进行相关的改造, 使它能够按需要正确输出自定义类型数据, 这在后面的章节中会详细讲到。

最后一行语句的输出结果是 The number of fdata is 34.343423, 为什么不是 The number of fdata is 34.343423433222 呢? 这是因为没有规定要输出数据的格式, 默认情况下, 小数部分取 6 位。C++ 提供了一些输入输出控制符, 使用它可以改变输入输出对象的某些控制属性, 以达到改变如输出的有效数字位数、小数位数等格式的效果。

而当程序需要用户输入数据时, 可以使用 cin 对象, 格式如下。

cin>>变量; // ">>" 为流提取操作符

例如:

```
void main()
{
    double ddata;
    cout<<"Please input a double data"<<endl;
    cin>>ddata;
    cout<<"The data you input is "<<ddata<<endl;
}
```

运行程序, 首先屏幕会输出 "Please input a double data", 输入 21.333, 然后屏幕就会输出 21.333, 如下所示(>>代表输入内容):

```
Please input a double data
>>21.333
21.333
```

其他类型的变量输入与之类似。

2. 格式化输入输出

虽然 C++ 提供了 cout 和 cin 进行输入和输出, 但是依然有很多程序员习惯于使用传统的 C 语言输入输出方式, 而且在分析前人写的某些 C 语言程序时更是会大量遇到, 因此有必要给大家对 C 语言的输入输出做一个简单介绍。最常用的输入输出函数有格式化输入函数 scanf() 和格式化输出函数 printf()。

1) scanf() 函数

scanf() 函数的一般格式如下。

scanf("控制字符串", 输入项列表);

其中控制字符串规定数据的输入格式, 必须用双引号括起, 其内容是由格式说明和普通字符两部分组成。输入项列表则由一个或多个变量地址组成, 当变量地址有多个时, 各变量地址之间用逗号 “,” 分隔。

scanf() 函数扫描获取用户的输入, 并将这些输入值保存到参数表内相应的变量地址中。

因此参数表中提供的一定是变量地址，就是变量名前加“&”。“&”叫做取地址操作符，作用是取得操作数(变量)的地址。而且应注意输入类型与变量类型要一致。

控制字符串由两个部分组成：格式说明和普通字符。

(1) 格式说明。

格式说明规定了输入项中的变量以何种类型的数据格式被输入，形式如下。

% [<修饰符>] <格式字> // "[]"代表该项为可选参数

不同类型的数据对应不同的格式字，详见表 3-1。

表 3-1 格式字符及其含义

格式字符	含 义
d	输入一个十进制整数
o	输入一个八进制整数
x	输入一个十六进制整数
f	输入一个小数形式的浮点数
e	输入一个指数形式的浮点数
c	输入一个字符
s	输入一个字符串

修饰符部分用于进行格式化输入，是可选项，包括以下 3 个。

- 字段宽度修饰符。

例如：

```
scanf ("%3d"&a);
```

按宽度 3 输入一个整数赋给变量 a。

- “l” 和 “h” 修饰符。

可以和 d、o、x 一起使用，加 l 表示输入数据为长整数，加 h 表示输入数据为短整数。例如：

```
scanf ("%10ld%hd ", &x, &i);
```

则 x 按宽度为 10 的长整数型读入，而 i 按短整数型读入。

- 字符*。*表示按规定格式输入但不赋予相应变量的作用，作用是跳过相应的数据。

例如：

```
scanf ("%4d*d%4d ", &x, &y, &z);
```

执行该语句，若输入为“1 2 3”，则结果为 $x=1$ ， $z=3$ ， y 未赋值，2 被跳过。

(2) 普通字符。

普通字符包括空格、转义字符和可打印字符。

2) printf()函数。

printf()函数一般格式如下。

```
printf ( "控制字符串", 输出项列表);
```

printf()函数中的控制字符串定义原则与 scanf()函数完全相同，这里不再赘述，输出项列表为变量列表，无须加取地址操作符。

例如：

```
#include <stdio.h>
void main()
{
    int i;
    printf("请输入一个整数:");
    scanf("%d",&i);
    printf("您输入的整数为: %d",i);
}
```

运行程序，得：

请输入一个整数

>> 23

您输入的整数为：23

3.2 字符集和关键字

C++作为一门程序设计语言，与人们日常生活中讲的自然语言(汉语、英语等)有很多相似之处。每种语言都是由一些基本的字符(或字)组成的，有了这些字符(字)，才能进而组词、造句，写出华美的文章。所有这些字符组成的集合称为这种语言的字符集。C++作为语言的一种(虽然它是用来和计算机打交道的)，也是由基本的字符集组成的，具体如下。

- 26 个大写英文字母：A~Z。
- 26 个小写英文字母：a~z。
- 数字字符：0~9。
- 特殊字符：空格，!，#，%，^，&，*，_(下划线)，+，=，-，~，<，>，/，\，'，"，;，，，，()，[]和{}。

有了这些字符以后,便可以开始组词了。C++中已经定义好了一些这样的词,就像人类的语言中已经固有的那些诸如行走、奔跑之类的词一样。在程序设计语言中,称这些词为关键字(有些书上也称保留字,因为这些是设计 C++语言时定义好的,具有特殊的意义)。C++中预定义的关键字见表 3-2。

表 3-2 C++中预定义的关键字

asm	auto	bad_cast	bad_typeid
bool	break	case	catch
char	class	const	const_cast
continue	default	delete	do
double	dynamic_cast	else	enum
except	explicit	extern	false
finally	float	for	friend
goto	if	inline	int
long	mutable	namespace	new
operator	private	protected	public
register	reinterpret_cast	return	short
signed	sizeof	static	static_cast
struct	switch	template	this
throw	true	try	type_info
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	while		

关键字由 C++系统定义,每个关键字都有其特殊含义,不允许重新作为其他名称使用。针对不同的编译环境,各编译器生产厂家还定义了一些针对该编译器的关键字,当在这些环境下编写程序时,这些扩充的关键字同标准 C++关键字一样不能被重新赋予其他含义。C++中扩充的关键字如表 3-3 所示。

表 3-3 C++中扩充的关键字

Allocate	inline	property
asm	int8	selectany
based	int16	single_inheritance
cdecl	int32	stdcall

续表

<code>declspec</code>	<code>int64</code>	<code>thread</code>
<code>dllexport</code>	<code>leave</code>	<code>try</code>
<code>dllimport</code>	<code>multiple_inheritance</code>	<code>uuid</code>
<code>except</code>	<code>naked</code>	<code>uuidof</code>
<code>fastcall</code>	<code>nothrow</code>	<code>virtual_inheritance</code>
<code>finally</code>		

除了这些预定义的关键字外,其他任何程序员创造的名字都称标识符(注:关键字也是标识符的一种),在C++中定义标识符需满足一定的规则。

标识符的构成规则如下。

- 以大写字母、小写字母或下划线(`_`)开始。
- 可以由以大写字母、小写字母、下划线(`_`)或数字0~9组成。
- 大写字母和小写字母代表不同的标识符。

例如: `Myname`、`_ASDFASD`、`b2423443`、`_as343afdas3_`都是合法的标识符,但下面的标识符是不合法的。

`3sdfas`: 违反了第一条法则,标识符只能以大写字母、小写字母或下划线(`_`)开始。

`cysee*asdf`: 违反了第二条法则,标识符声明中包含非法字符“*”。

`union`: 关键字,不能当标识符使用。

3.3 C++的数据类型概述

编写程序的目的即通过一定的算法,说通俗一点就是通过一系列的操作,对数据进行输入、存储、加工、处理,最终以某种方式输出给用户。即“程序=算法+数据结构”。

因此可以说,数据是一切工作的核心,所有的算法都围绕数据展开,没有数据的程序不是程序。因此,学习如何正确地定义数据是处理这些数据、实现特定功能的基础。

在定义数据类型之前,首先要确定要描述的数据是哪种类型。数据类型定义了数据的存储空间,定义了数据的访问方式和适用范围。根据不同的分类方式,数据类型也有多种不同的划分方式。

根据数据在程序执行期间是否可以被改变,可以将数据分为变量和常量。顾名思义,变量即在程序运行期间数据值可以被赋予新的数值,而常量则是一经定义的不允许再更改的数据。

根据数据类型的定义则可分为基本数据类型和用户自定义数据类型。基本数据类型由系统预先定义，这种类型的数据在几乎所有的程序中都会用到。用户自定义数据类型是用户为满足特定的需要，通过基本数据类型的组合而成的新的数据类型，这种类型通常只针对特定的某一类应用程序。图 3-1 说明了基本数据类型和非基本数据类型包含的种类。

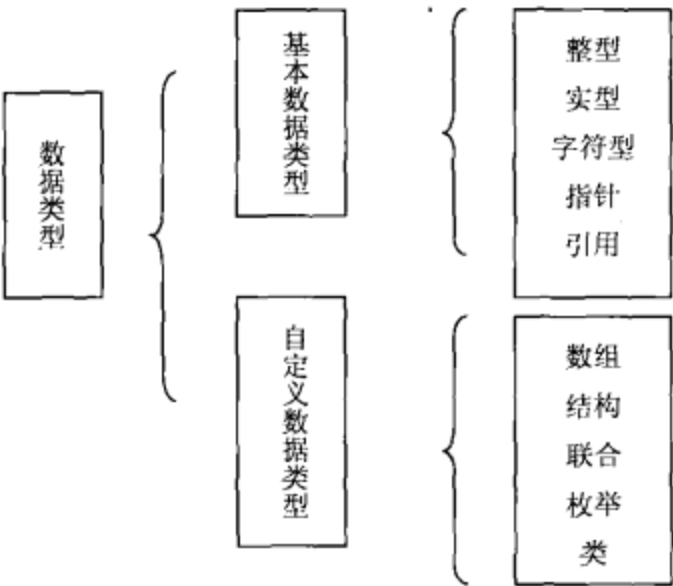


图 3-1 数据类型分类图

对于某些基本数据类型，根据其存储空间的大小不同还可以再细分，比如：整型数又可分为短整型：short int；普通整型：int；长整型：long int。由于存储空间大小不同，不同的数据类型表示数据的范围和精度也各异。举个最简单的例子：在 16 位编译器上，int 占用的存储空间为 16 位，能表示的数据范围-32 768~32 767；long int 占用 32 位存储空间，故能表示的范围为-2 147 483 648~2 147 483 647。根据数据是否有正负之分，还可以分为有符号的 signed 和无符号的 unsigned。表 3-4 列出了 C++的基本数据类型的描述及占用空间。

表 3-4 C++的基本数据类型及占用空间

数据类型	占用空间
char, unsigned char, signed char	1 byte
short, unsigned short	2 bytes
int, unsigned int	4 bytes
long, unsigned long	4 bytes
float	4 bytes
double	8 bytes
long double	8 bytes

注：表中列出的长度和范围为 32 位机器上的结果。

在计算机中，数据以字节为基本存储单元，1 个字节(byte)为 8 个二进制位，在不同的机器上各种数据类型的长度是不同的。比如：int 型的存储定义为一个机器字长，在 32 位编译器上它的长度为 4 个字节，但当程序移植到 64 位编译器上时，它的长度就变为 8 个字节。一般说来，在短字长机器上开发的程序拿到长字长机器上运行不会有什么影响。但是，当将长字长机器上的程序拿到短字长机器上运行的，就有可能因为字长的不同而使程序无法正常运行。

为了使程序能够在不同种类的机器上都能正常运行，有些时候，不得不要先取得每种数据类型所表示的长度。在 C++ 中，可以通过使用操作符 `sizeof` 来完成长度的计算。例：

```
cout<<"The Length of int is "<<sizeof(int)<<endl;
```

在 32 位机器上输出为：

```
The Length of int is 4
```

3.4 基本数据类型

C++ 语言中常用的基本类型有：整型、浮点型、字符型、布尔型、空类型等。

3.4.1 整型数据

C++ 中的整型数据与通常所说的整数对应，整型数据用关键字 `int` 定义。根据表示方式的不同可以分为十进制、八进制和十六进制。通常在日常生活中多采用十进制表示数据，但在计算机中的数据都是以二进制的形式存储的，编写程序时，有时候用八进制或十六进制表示数据会更方便。比如为了描述一幅二值图像中的八个点，假设前四个为黑色、后四个为白色时，用二进制可表示为“00001111”，用十六进制可表示为“0F”，显得非常直观；但如果用十进制则为“15”，便会让人觉得摸不着头脑。因此，在编写程序时，选用恰当的表达方式会给开发带来很多方便。

十进制的表示方法和日常中采用的数字表示方法相同，使用 0~9 十个字符的排列和组合表示数字。

八进制使用 0~7 八个字符的排列和组合表示数字，每逢八向前进一位，在代码中使用八进制的数字时还需要在要表示的数字前加 0，以和十进制相区别。

十六进制使用 0~9 和 a~f 十六个字符的排列和组合表示数字，每逢十六向前进一位，在表示上加 0x 作为前缀。下面是三种表示方式的例子。

- 124: 十进制。
- 0334: 八进制, 在数字前加 0 作为标识。
- 0xff: 十六进制, 在数字前加 0x 作为标识。

定义整型数据变量的一般格式如下。

```
int 标识符[=数值]
```

下面是使用整型数据的例子:

```
void main()
{
    int i=0,j=100;
    cout<<"The value of number i is "<<i;
    cout<<"The value of number j is "<<j;
}
```

3.4.2 浮点型数据

通常程序中使用的数据并不都是整数, 因此必须有一种数据类型来描述整数以外的数据, 这便是浮点型数据。浮点型也被称作实型。浮点型的数值包括整数部分、尾数部分和指数部分。浮点型常量的形式如下。

[整数] [.尾数] [E|e[+|-]指数]

小数点前为整数部分, 小数点后为尾数部分。E(e)为指数符号, 代表后面的部分为数的指数部分, 指数部分只是在使用科学技术法表示数值的时候才会用到, 多数情况下此部分会被省略掉。

指数可以为正也可以为负, 当为正数时前面的正号可以省略; 但当指数为负数时, 必须加上负号。需注意指数必须为整数。

根据数据表示范围的大小, 浮点型数据可以分为 float(单精度)、double(双精度)和 long double(长双精度)三种。在 C++中, 默认的浮点型数据为双精度。当需要用到 float 型数据时, 需要预先声明或是在数字后面加上 f 或 F 作为 float 型数据的后缀, 以 l 或 L 作为 long double 型数据的后缀。

下面是一些典型的浮点型数据的实例。

- 1.5f: float 型。
- 0.8: double 型。
- 3.45e-8: 指数形式, double 型。

当浮点型数据的整数部分为 0 时, 可以省略, 如上面例子中的 0.8 可以写为 .8。但是指

数形式的指数前面必须有数字，如 e3、e15 这些都是非法的浮点数。

注意，浮点型变量是有精度限制的，当程序给出的数字长度超过它的数据类型的最大长度时，后面的数字将自动被截断。如：

```
float a = 14.43523543543542354;
```

由于 float 类型的精度为 7 位有效位，14.435235 后面的数据会被系统截断，因此这部分的内容在程序中实际上是无效的。那么，如果对数据的精度要求非常高，例如处理某些地形数据时，就需要将这些数据声明为 double 或 long double 类型，以满足精度的要求。

3.4.3 字符型数据

1. 字符常量

字符常量是指用一对单引号括起来的一个字符，如'a'、'9'和'!'等。字符常量中的单引号起定界作用，并不表示字符本身。单引号中的字符不能是单引号(')和反斜杠(\)，它们特有的表示法将会在转义字符中介绍。

在 C 语言中，字符是按其所对应的 ASCII 码值来存储的。ASCII 码实际上就是一个八位的二进制整数，一个数字代表一个字符，一个字符占一个字节，举例如表 3-5 所示。

表 3-5 几个字符及对应的 ASCII 码值

字 符	ASCII 码值(十进制)
!	33
0	48
1	49
9	57
A	65
B	66
a	97
b	98

注意字符'9'和数字 9 的区别，前者是字符常量，后者是整型常量，它们的含义和在计算机中的存储方式都截然不同。

由于 C 语言中字符常量是按短整型数(short 型)存储的，所以字符常量可以像整数一样在程序中参与相关的运算。例如：

```
'a'-32;           //相当于执行 97-32，结果为 65
```

```
'A'+32;           //相当于执行 65+32, 结果为 97
'9'-9;           //相当于执行 57-9, 结果为 48
```

2. 字符串常量

字符串常量是指用一对双引号括起来的一串字符。双引号只起定界作用, 双引号括起的字符串中不能是双引号(")和反斜杠(\), 它们特有的表示法在转义字符中介绍。例如: "China", "Cprogram", "YES&NO", "33312-2341", "A"等。

字符串常量在内存中存储时, 系统自动在字符串的末尾加一个“串结束标志”, 即代表空(NULL)的字符"\0", ASCII 码值为 0。因此在程序中, 长度为 n 个字符的字符串常量, 在内存中占有 n+1 个字节的存储空间。

例如, 字符串 China 有 5 个字符, 作为字符串常量"China"存储于内存中时, 共占 6 个字节, 系统自动在后面加上"\0"字符, 其存储形式如下。

C	h	i	n	a	\0
---	---	---	---	---	----

要特别注意字符常量与字符串常量的区别, 除了表示形式不同外, 其存储性质也不相同, 字符'A'只占一个字节, 而字符串常量"A"占两个字节。

3. 转义字符

转义字符是 C 语言中表示字符的一种特殊形式。通常使用转义字符表示 ASCII 码字符集中不可打印的控制字符和具有特定功能的字符, 如用于表示字符常量的单引号('), 用于表示字符串常量的双引号(")和反斜杠(\)等。转义字符用反斜杠后面跟一个字符或一个八进制或十六进制数表示。表 3-6 给出了 C 语言中常用的转义字符。

表 3-6 转义字符的含义及对应的 ASCII 码值

转义字符	意 义	ASCII 码值(十进制)
\a	响铃(BEL)	007
\b	退格(BS)	008
\f	换页(FF)	012
\n	换行(LF)	010
\r	回车(CR)	013
\t	水平制表(HT)	009
\v	垂直制表(VT)	011
\\	反斜杠	092
\?	问号字符	063

续表

转义字符	意 义	ASCII 码值(十进制)
\'	单引号字符	039
\"	双引号字符	034
\0	空字符(NULL)	000
\ddd	任意字符(用三位八进制数表示)	
\xhh	任意字符(用二位十六进制数表示)	

字符常量中使用单引号和反斜杠以及字符常量中使用双引号和反斜杠时,都必须使用转义字符表示,即在这些字符前加上反斜杠。

在 C 程序中使用转义字符"\ddd"或者"\xhh"可以方便灵活地表示任意字符。"\ddd"为斜杠后面跟三位八进制数,该三位八进制数的值即为对应字符的八进制 ASCII 码值。"\x"后面跟两位十六进制数,该两位十六进制数为对应字符的十六进制 ASCII 码值。

使用转义字符时需要注意以下几个问题。

- 转义字符中只能使用小写字母,每个转义字符只能看作一个字符。
- "\v"(垂直制表)和"\f"(换页符)对屏幕没有任何影响,但会影响打印机执行响应操作。
- 在 C 程序中,使用不可打印字符时,通常用转义字符表示。

4. 字符变量

字符变量用来存放字符常量,注意只能存放一个字符,在字符变量中不可以存放字符串。

字符变量的定义形式如下。

```
char c1,c2;
```

它表示 c1 和 c2 为字符变量,各放一个字符。因此可以用下面语句对 c1、c2 赋值。

```
c1='a';
c2='b';
```

由于每个字符与唯一的 ASCII 码值对应,所以字符变量可以看作一个整型变量,且可以用整数方法来操作字符变量。例如:

```
main()
{
    char c1,c2;
    c1=97; c2=98;
    printf("%c %c",c1,c2);
}
```

c1、c2 被指定为字符变量。但在第 3 行中，将整数 97 和 98 分别赋给 c1 和 c2，它的作用相当于以下两个赋值语句。

```
c1='a'; c2='b';
```

因为'a'和'b'的 ASCII 码值为 97 和 98，故可如上赋值。第 4 行表示输出两个字符，"%c" 是输出字符的格式。程序输出为：a b。

再例如：

```
main()
{
    char c1,c2;
    c1='a'; c2='b';
    c1=c1-32; c2=c2-32;
    printf("%c %c", c1,c2);
}
```

运行结果为：A B

上述程序的功能是将两个小写字母转换为大写字母。因为'a'的 ASCII 码值为 97，而'A'为 65，'b'为 98，'B'为 66。从 ASCII 代码表中可以看出每一个小写字母比大写字母的 ASCII 码值大 32，即'a'='A'+32。

3.4.4 bool 类型

bool 类型^①(即有些语言中的布尔型)用来描述真和假两种数据，真用 true 表示，假用 false 表示。bool 型变量多用于条件语句作为逻辑判断的结果，如下例。

```
void main()
{
    int var1,var2;
    bool isbig;
    var1 = 1;
    var2 = 3;
    isbig=( var1> var2);
    if(isbig)
    {
        cout<<"var1 比 var2 大"<<endl;
    }
    Else
```

① 在传统的标准 C 语言中是没有 bool 类型的，这部分是 C++新增加的内容。

```
{
    cout<<" var1 比 var2 小"<<endl;
}
}
```

因为 var1 的等级小于 var2, 所以 isbig 取 false(假), 故程序输出结果为:

var1 比 var2 小

bool 类型的另一种常见用途是作为函数的返回值, 当一个函数完成一项指定的功能(有关函数的概念请参见第 8 章 8.1 节), 需要判断这项功能是否被正确完成时, 通常以 bool 型的返回值作为判断的标志。

```
bool writetofile()
{
    File *fp;
    if(!(fp=(fopen("test.txt", "w"))))①
    {
        return false;
    }

    fprintf(fp, "测试数据");

    fclose(fp);

    return true;
}

void main()
{
    if(writetofile())②
    {
        cout<<"写文件失败"<<endl;
    }
}
```

- ① 在 C++ 中规定, bool 值为 true 的值为 1, false 的值为 0, 但是, 当用于逻辑判断时, 任意非 0 值都可以视为真。所以类似于下面例子的情形是 C++ 中程序的常见写法。

```
int a=1,b=2;
if(a+b)
{
    cout<<"true"<<endl;
}
```

因为 a+b 的结果是 3, 属于非 0 值, 所以在逻辑上表示真, 所以程序会输出 true。

- ② 游戏中的很多配置数据和资源都是存储在相应的文件中的, 对这些文件的操作是很频繁的事情, 建议对每次这样的操作都判断一下函数是否成功执行, 当没有完成指定的操作时, 程序应进行相应的处理工作, 否则应用程序便有可能在执行途中莫名其妙的崩溃。有关文件操作请参见第 17 章 17.5 节。

```
    else
    {
        cout<<"存储成功"<<endl;
    }
}
```

在这个例子中，先创建了一个名为 test.txt 的文件，然后试图向其中写入“测试数据”。当不能创建文件时，函数 writetofile() 会返回 false 值，程序输出“写文件失败”；若数据被成功写入文件，函数 writetofile() 会返回 true 值，程序输出“存储成功”。

3.4.5 void 类型

严格来说，void 并不是真正的数据类型，只是因为语法的完备性需要而定义的一个基本数据类型。它可以作为更复杂数据类型的组成部分，但是不存在任何类型为 void 的变量或对象(见面向对象部分)。

在程序中，void 的用途有以下两种。

- 用于指明函数没有返回值。

例如：

```
void func();
void main();①
```

- 使用指针时，若不明指针的数据类型，一律以 void 表示。

例如：

```
void * pointer;②
```

3.4.6 常量与变量

前面已经介绍了 C++ 程序中使用的数据有常量与变量之分。所谓常量，是指在程序设计阶段，数据的值已被确定，程序运行期间不可更改的这部分数据。常量的值是在设计阶段即被指定的，在程序运行阶段，常量被存储在特定的存储空间中，任何对这部分数据的修改都是非法的。常量可以是任何数据类型，举例如表 3-7 所示。

① 省略函数的返回值不代表函数不返回值，默认情况下函数的 C 语言[0]返回值类型为 int 型，但 C++ 不允许用默认的类型，因此，C++ 的函数必须明确函数类型。

② 有关指针的细节内容详见指针部分。

表 3-7 数据类型及对应的常量举例

数据类型	常量举例
char	'a'、'\n'、'9'
int	21、123、2100、-234
long int	35000、-34
short int	10、-15、90
unsigned int	10000、987、40000
float	123.23、4.34e-3
double	12312333、0.9876234

定义常量的方法有以下两种。

1. 使用宏#define 定义常量

定义的一般格式为：

#define 标识符值

这也是传统的 C 语言宏定义常量的方式，比如按照上述方式定义圆周率为常量 PI：

```
#define PI 3.1415926①
```

特别的，C++还支持字符串常量的定义。例如：

```
#include <iostream>

using namespace std;

//定义符号常量
#define MACROSTRING "这是一个自定义的宏!"

void main()
{
    //输出符号常量的内容
```

① 根据 C++字符集的定义可知， π 不属于基本字符集的字符，所以在程序中不能应用该字符，因此要用其他的符号代替，相当于起了一个其他的名字一样，这在编写程序时是很常见的现象，尤其是当需要用中文名字作为标识符时更是不允许的(有数据库设计经验的读者可能会在此感觉不太舒服，因为现在使用的数据库中有些是支持中文名的)。尽量在应用程序中使用英文单词或缩写作为标识符是一个很好的习惯，当然这不是说用别的命名规则不可以，只是这种命名规则是大多数程序员都认可的风格，而且容易表明意义。在程序的命名规则上保持与其他人一致会使项目进度非常顺利，比如本例中的 PI，就是一个比较通用的表示。在本书后面的章节中会继续介绍这方面的内容，不在此赘述。

```
    cout<<MACROSTRING<<endl;
}
```

在本例中定义了一个字符串常量 `MACROSTRING`，以后在程序中需要使用这个字符串常量的时候都可以直接使用符号 `MACROSTRING`。该程序的运行结果为：

这是一个自定义的宏！

下面再看一个定义整型常量的例子：

```
#define WeekNum 7
```

这个例子将 `WeekNum` 定义为每周的天数 7，以后需要用到每周的天数的时候就可以直接使用 `WeekNum` 了。

用宏定义常量有一个严重的缺陷：在进入编译器之前宏会被预处理程序用该宏代表的内容所替换掉(关于编译预处理的内容请参阅本书第 6 章的内容)，也就是说在编译之前程序中所有的 `WeekNum` 将被 7 代替，`WeekNum` 不会加入到符号列表中。如果涉及这个常量的代码在编译时报错，就会令人很费解，因为报错信息指的是 7，而不是 `WeekNum`。如果 `WeekNum` 不是在用户写的头文件中定义的，分析程序的工程师就会奇怪 7 是从哪里来的，甚至会花时间跟踪下去。这个问题也会出现在符号调试器中，因为同样的，符号名不会出现在符号列表中。

为了解决这个问题，C++同时还支持另一种定义常量的方式。

2. 使用 `const` 定义常量

定义的一般格式为：

```
const 数据类型 标识符=值
```

例如：

```
const int MAX = 100;           //C++ 语言的 const 常量
const float PI = 3.14159;      //C++ 语言的 const 常量
```

在 C++ 中，用 `const` 来定义常量比用 `#define` 来定义常量有更多的优点。

- `const` 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误(边际效应)。所谓边际效应举例如下。

```
#define N 100
#define M 200 + N
```

当程序中使用 `M*N` 时，原本想要 `100 * (200 + N)` 的却变成了 `100 * 200 + N`。

- 有些集成化的调试工具可以对 `const` 常量进行调试，但是不能对宏常量进行调试。
- 在 C++ 程序中只应使用 `const` 常量而不使用宏常量，即 `const` 常量完全取代宏常量。

在常量定义上一般有如下的习惯规则。

- 需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为便于管理，可以把不同模块的常量集中存放在一个公共的头文件中。
- 如果某一常量与其他常量密切相关，应在定义中表现出这种关系，而不应给出一些孤立的值。例如：

```
const float RADIUS = 100;
const float DIAMETER = RADIUS * 2;
```

与常量对应的是变量，即在程序运行期间可以改变的数据。同常量一样，变量也占用一定的存储空间，有自己的内存地址，但是具体存储位置因变量的类型而异。由于变量的值可以在程序运行期间改变，所以变量在声明时既可以预先将其初始化为某个具体的值，也可以省略，当程序需要用到这些变量时再给它们赋予相应的值。变量的定义格式如下。

数据类型 标识符; //声明单个变量

或用逗号分割，同时声明同一类型的多个变量，定义格式如下。

数据类型 标识符 1, 标识符 2, ..., 标识符 n; //声明多个变量

其中，标识符的定义必须符合 C++ 标识符定义的规定，且不能与预定义的关键字冲突。

例如：

```
int i, j=0;
```

在上例中，同时声明定义了两个变量。变量 `i` 声明过程中所做的只是分配存储空间，并没有进行初始化；`j` 在声明时进行了初始化。所谓变量的初始化，是指当为变量分配存储空间时，同时为变量赋予特定的初值，这在养成良好的编程风格、增加代码的健壮性方面很重要，在后面讲到指针、数组和类的时候，还会进一步强调这方面的内容。

3.5 类型转换

不同类型的数值型数据进行混合运算时，必须要先转换成同一类型之后再运算。^①

① 严格说来，本节内容应该在运算符与表达式中进行说明，但是，本章是介绍数据类型的，在实际使用的时候，一旦涉及数据类型，就可能会牵涉到数据类型的转换，因此特意将本节内容放在这里。

3.5.1 隐式类型转换

C++定义了一组内置类型对象之间的标准转换，在必要时它们被编译器隐式地应用到对象上。这种转换是系统自动进行的，因此被称为隐式类型转换。

隐式类型转换规则如下。

- 为防止精度损失，如果必要的话，精度低的类型总是被提升为精度高的类型。
- 所有含有小于整型的有序类型的算术表达式在计算之前，其类型都会被转换成整型。也就是说，char 型和 short 型在参与运算时，必须先转换成 int 型。
- 所有的浮点运算都是以双精度进行的，即使仅含 float 单精度量运算的表达式，也要先转换成 double 型，再进行运算。

图 3-2 表示了不同数据类型间进行运算时的隐式类型转换方向。其中，float 型向 double 型的转换和 char 型向 int 型的转换是必定要进行的，即不管另一个运算对象是否为不同的类型，这种转换都要进行。

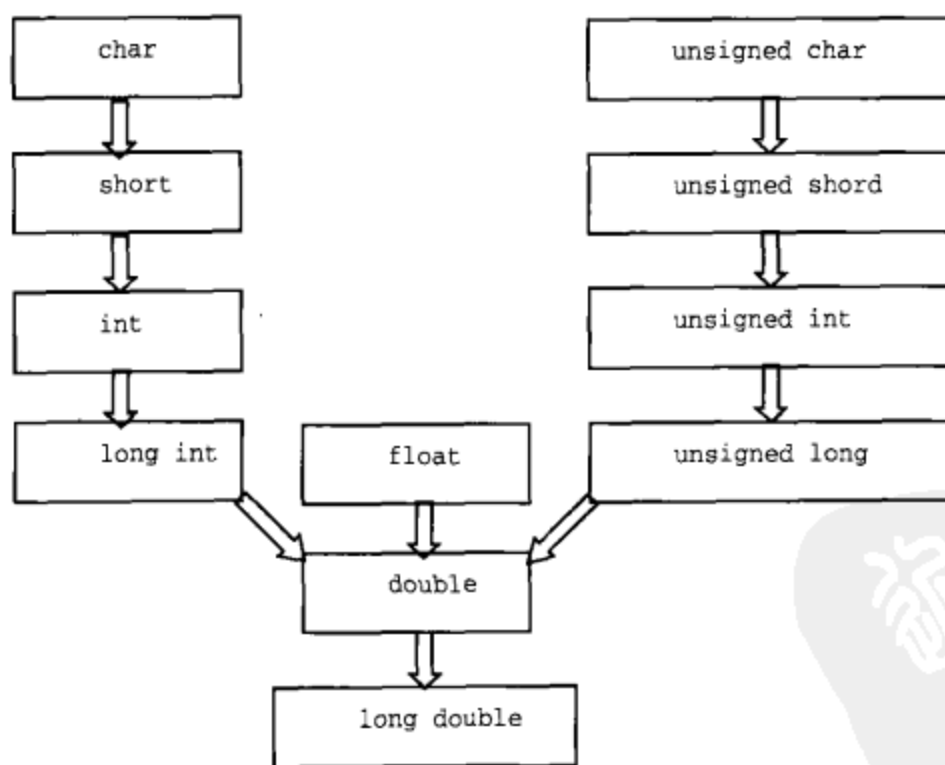


图 3-2 类型转换方向图

纵向箭头表示当运算对象为不同类型时的转换方向。如 int 型与 double 型数据进行运算时，是先将 int 型转换为 double 型，再对 double 型数据进行运算，最后的运算结果也为 double 型。

例如， $100 - 'a' + 40.5$ 这个表达式的运算过程是：第一步，计算 $100 - 'a'$ ，先将字符 'a' 转换为整型数 97 (a 的 ASCII 码值)，运算结果为 3；第二步，计算 $3 + 40.5$ ，先将 float 型的 40.5

转换为 double 型，再将 int 型的 3 转换为 double 型，最后的运算结果为 double 型。

隐式类型转换一般发生在下列这些典型的情况下。

(1) 算术转换：在混合类型的算术表达式中，是最为常见的。在这种情况下，最宽的数据类型成为目标转换类型，这也被称为算术转换。例如：

```
int type_int=3;
double type_double=3.14159;
type_int+type_double;    //type_int 被隐式转换为 double 类型 3.0
```

(2) 赋值转换：用一种类型的表达式赋值给另一种类型的对象，在这种情况下目标转换类型是被赋值对象的类型。例如针对上例，下面第一个赋值中 double 型的值被截取成 int 型的值。

```
type_int=type_double;    //type_double 被截取为 type_int 值 3
```

如果右边的数据类型长度比左边长时，将丢失一部分数据，这样会降低精度，例如：

```
#include <iostream>
using namespace std;

void main()
{
    float PI=3.14159;
    int s,r=5;
    s=r*r*PI;
    printf("s=%d\n",s);
}
```

程序运行结果如下：

```
s=78
```

本例程序中，PI 为实型；s、r 为整型。在执行 `s=r*r*PI` 语句时，r 和 PI 都转换成 double 型计算，结果也为 double 型。但由于 s 为整型，故赋值结果仍为整型，舍去了小数部分。

- 把一个表达式作为参数直接传递给一个函数调用。若表达式的值的类型与对应形式参数的类型不相同，在这种情况下目标转换类型是形式参数的类型，例如：

```
void function(double);
function(2)    //用整型数 2 来调用，转换为 double 类型的 2.0
```

- 从一个函数返回一个表达式。若表达式的类型与返回类型不相同，在这种情况下目标转换类型是函数的返回类型。

3.5.2 强制类型转换

利用强制类型转换运算符可以将一个表达式转换成所需的类型。强制类型转换的一般形式是：

(类型名)表达式

或者

类型名(表达式)

例如：

(double)a: 将 a 转换成 double 型。

(int)(x+y): 将 x+y 的值转换成 int 型(注意，不能写成“(int)x+y”)。

强制类型转换一般用于自动类型转换不能达到目的的时候。例如，sum 和 n 是两个 int 型变量，则 sum/n 的结果是一个舍去了小数部分的整型数，这个整数很可能存在较大的误差，如果想得到较为精确的结果，则可将 sum/n 改写为 sum/(float)n 或(float)sum/n。

在使用强制转换时应注意以下问题。

类型说明符和表达式都必须加括号(单个变量可以不加括号)，如把(int)(x+y)写成(int)x+y，则成了把 x 转换成 int 型之后再与 y 相加了。

无论是强制转换还是自动转换，都只是为了本次运算的需要而对变量的数据长度进行临时性转换，而不改变数据说明时对该变量定义的类型。

本章小结

C++的基本数据类型可在广义上分为整型和实型两大类。广义整型包括字符型(一个字节)、短整型(两字节的整型)、整型(两字节或四字节的整型)和长整型(四字节的整型)，它们可分为有符号的(signed)和无符号的(unsigned)；实型数据包括单精度浮点型(float，占四字节)、双精度浮点型(double，占八字节)和长双精度浮点型(long double)。

布尔型可能在某些 C++编译器中无定义，用户可自行定义，或者用字符型或整型代替。

定义各种基本数据类型的作用是为了在定义变量时，在编译或运行时在数据区或堆栈区分配已知长度的内存。

变量说明时必须指出类型，变量应该赋初值后使用；常量标识符的说明要在前面加 `const`。

本章还介绍了标识符和关键字，标识符必须以字母或下划线开头，中间可以用数字，关键字是特殊的标识符，用户定义的标识符不能与关键字相同。



第 4 章 运算符与表达式

本章内容：

- 运算符。
- 表达式。
- 强制转换。

重点：

运算符与表达式。

目的：

利用基本的运算符和表达式，产生基本的程序语句，进行简单的数据运算和简单的程序书写。

4.1 概 述

程序设计是对运算语义的表达，其基本功能就是告诉计算机通过何种运算得到何种结果。在本章中，将详细讲述 C++ 语言中的运算符和表达式的使用方法。

运算是计算机语言要实现的最基本的功能，由运算符和变量、常量等组成的运算式称做表达式。各种程序功能的实现都建立在各种运算的基础之上。

4.2 运算符和表达式

4.2.1 运算符和表达式的种类

表达式是由函数、变量、运算符、数字、字符串。常数等组成的式子，是对一组数据

进行计算、赋值或进行逻辑判断等运算作描述的基本单元。运算符^①是用以表明运算类型的符号，C++的运算符包括以下几种。

- 算术运算符。
- 关系运算符。
- 逻辑运算符。
- 赋值运算符。
- 位运算符。
- 条件运算符。
- 逗号运算符。
- sizeof 求字节运算符。
- 强制类型转换运算符。
- 指针与取地址运算符。
- 内存分配与释放操作符。
- 分量运算符。
- 下标运算符。
- 其他。

根据运算性质的不同，表达式也可以分为以下几种。

- 算术表达式。
- 关系表达式。
- 逻辑表达式。
- 赋值表达式。
- 位运算表达式。
- 条件运算表达式。
- 逗号表达式。
- 其他。

4.2.2 左值和右值

在程序设计中，左值与右值是一个基本问题。任何表达式都有左值和右值之分。简单地说，左值是能够出现在赋值表达式左边的表达式；右值只能出现在赋值表达式的右边。注意上面所说的“能够”和“只能”的区别，也就是说，左值表达式也可以作为右值表达

^① 某一种符号在不同的情况下可能作为不同的运算符使用，如()可以用于进行强制类型转换，也可以用来表示函数调用。在第14章还将看到运算符被重载的情况。

式，而右值表达式永远不可能成为左值表达式。

从内存的角度看，左值是变量所代表的内容在内存中存放的地址，它拥有存放数据的空间，因此可以被赋值；右值则是该变量代表的内容。图 4-1 形象地给出了左值与右值的关系。

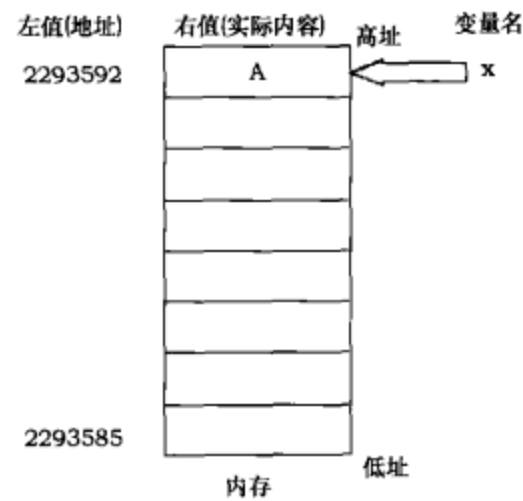


图 4-1 左值与右值的关系

在图 4-1 中，变量 x 代表存放在内存中的一个字符'A'，该字符便是变量 x 的右值，其左值如图 4-1 所示。

4.3 算术运算符和算术表达式

C++中的算术运算符包括双目的加减乘除四则运算符和求模运算符，以及单目的正负运算符，如表 4-1 所示。

表 4-1 算术运算符及其含义

算术运算符	含 义
+	加法
-	减法(双目)/求负(单目)
*	乘法
/	除法
%	取模

+、-、*、/四种运算符的操作数，可以是任意基本数据类型。其中+、-、*与一般算术运算规则完全相同。减号在用于单目运算时实际上是取变量的相反数，即若变量 a 的值为 7，则-a 的运算结果为-7。

当整数相除时，运算的结果仍为整数，结果的小数部分将被丢弃。如：15/7 的结果为2。但是，当结果为负数，即被除数与除数有且只有一个为负数时，不同的编译器给出的结果是不同的。有些编译器的结果取不超过数学运算结果的最大整数，如-10/4 的数学运算结果为-2.5，程序给出的结果为-3。但是多数编译器采取的仍是丢弃小数部分的结果，Visual C++就是这样设计的。若被除数或除数有一个为浮点型，则运算浮点除法，结果包含小数部分，精度因数据类型的不同而异。%运算符的作用是用来求除法的余数，前后均应为整型数据。例如：

```
#include <iostream>
using namespace std;

void main()
{
    cout<<1/2<<endl;
    cout<<10/4<<endl;
    cout<<-7/5<<endl;
    cout<<-3/-4<<endl;
}
```

程序运行结果如下：

```
0
2
-1
0.75
```

用算术运算符和括号将运算对象连接起来的式子叫做算术表达式。例如：

```
a+2*b-5、18/3*(2.5+8)-'a'
```

同普通数学运算一样，在一个算术表达式中，允许不同的算术运算符以及不同类型的数据同时出现。因此在这样的混合运算中，就必须注意算术运算符的优先级。不仅是算术运算符，C++语言对每一种运算符都规定了优先级，在接下来将一一解释它们。

含有混合运算的表达式在求值时，应按次序从高优先级的运算执行到低优先级的运算。即按照先乘除后加减，求负数高于乘除，括号的优先级最高的顺序进行运算。当遇到同级运算时，按照从左到右的顺序进行处理。算术运算符的优先级从高到低排列为。() \rightarrow -(单目) \rightarrow * \rightarrow / \rightarrow % \rightarrow + \rightarrow -(双目)。

4.4 自增和自减运算符

当程序需要对变量做加1处理时，可以使用自增运算符“++”，减1时使用自减运算符

“--”，自增和自减运算符有如下两种形式。

```
++i;      --i;  
i++;      i--;
```

++i 和 i++ 单从运算结果上看都是使变量 i 加 1，即与 i=i+1 相同；但 ++i 是先将 i 加 1，然后取将加 1 后的变量^①作为表达式的值。例如，若 i=3，则 j=++i 的运算结果为 i=4，j=4，其运算过程等价于以下两句代码。

```
i=i+1;  
j=i;
```

而 i++ 是先取 i 的值作为表达式的值，以这个值进行操作，然后再将 i 的值加 1。例如，若 i=3，则 j=i++ 的运算结果为 i=4，j=3，其运算过程等价于以下两句代码。

```
j=i;  
i=i+1;
```

--i 和 i-- 的区别与 ++i 和 i++ 的区别相同，只是运算改为减 1 而已。

自增运算符和自减运算符是两个非常有用的运算符，它的执行效率比 i=i+1 或 i=i-1 更高，而且前者的写法使程序更精练。因而这两个运算符多用于循环表达式的循环条件加减 1 的操作，这在第 5 章会看到。

使用自增/自减运算符时需要注意以下几个方面。

- 自增/自减运算符仅可用于变量，不能用于常量或表达式。类似于 ++4 和 (i+j)++ 的写法都是不对的。这是因为自增/自减运算包括有改变原操作数的行为。
- ++ 和 -- 的结合方向是“自右至左”的。这点与普通算术运算的自左向右的结合性是不同的。例如：

```
#include <iostream>  
using namespace std;  
void main()  
{  
    int i=3,j=5;  
    cout<<i++<<j<<endl;  
    cout<<i<<endl;  
    cout<<j<<endl;  
}
```

程序运行结果如下：

① 这里一定要强调是变量而不是值，前++的返回结果是个引用，后++的返回结果才是值。

8
4
5

在这个程序执行时，根据自右至左的结合准则，“ $i+++j$ ”被理解为“ $i++ + j$ ”。因此先取 i 的原值进行 $i+j$ 运算，得到 8，然后再对 i 加 1 将其值改为 4。当然，像“ $i+++j$ ”这样的代码几乎完全没有可读性，除非特别需要，程序员一般不会使用这样的写法。

自增/自减运算符的优先级高于四则运算符，与求负运算符的优先级相同。

4.5 赋值运算符和赋值表达式

4.5.1 赋值运算符与赋值运算

在 C++ 中，用“ $=$ ”^①符号作为赋值运算符，赋值运算是双目运算，用来将一个数据赋给另一个变量。这里与常规的习惯不同，“ $=$ ”在 C++ 中没有等于的含义，因此要特别注意。在 4.5.2 节将看到，判断两个量相等时使用“ $==$ ”操作符。

赋值运算的形式有以下三种。

(1) 将常量值赋值给变量。

例如：

```
a=15;           //将常量 15 赋给变量 a
```

(2) 将变量值赋值给变量。

例如：

```
a=b;           //将变量 b 的值赋给变量 a
```

(3) 将表达式的结果赋值给变量。

例如：

```
a=b+c;         //将表达式 b+c 的值赋给变量 a
```

① 类似于算术运算，进行赋值运算时，若“ $=$ ”前后的数据类型不一致，则需要进行数据的类型转换，规则同有关数据类型转换的说明。

4.5.2 复合赋值运算符

在赋值符“=”之前加上其他运算符，即构成复合运算符。如在等号前加加号构成“+=”复合赋值运算符。复合赋值运算的意思是先将变量与后面的量进行前面运算符指定的运算，完成后再将运算结果赋给前面的变量，如： $i+=32$ 的意思为，先将 i 和 32 求和，然后将结果赋值给 i ，若初始 i 的值为 23，则进行复合运算后 i 的值变为 55。

赋值运算符与算术运算符结合构成的复合赋值运算符包括以下几个。

- +=运算符，例： $a+=3$ ；等价于 $a=a+3$ 。
- -=运算符，例： $a-=15$ ；等价于 $a=a-15$ 。
- *=运算符，例： $a*=15$ ；等价于 $a=a*15$ 。
- /=运算符，例： $a/=15$ ；等价于 $a=a/15$ 。
- %=运算符，例： $a%=15$ ；等价于 $a=a\%15$ 。

另外，还可以有位运算的复合赋值运算符<<=、>>=、&=、^=和|=。这些将在后面位运算中介绍。

采用复合赋值运算符，可以使程序的代码更加简洁，由于这种写法与“逆波兰式”^①一致，能提高编译效率，并产生质量较高的目标代码，所以在编写程序的时候，应尽可能地使用复合赋值运算符。

4.5.3 赋值表达式

用赋值运算符将一个变量和一个表达式连接起来，就成了赋值表达式。赋值表达式一般形式如下：

<变量名><赋值运算符><表达式>

即“变量 = 表达式”。

① 波兰科学家卢卡谢维奇(Lukasiewicz)提出的算术表达式的另一种表示，又称后缀表示。其定义是把运算符放在两个运算对象的后面。采用后缀表示的算术表达式被称为后缀算术表达式或后缀表达式。在后缀表达式中，不存在括号，也不存在优先级的差别，计算过程完全按照运算符出现的先后次序进行，整个计算过程仅需一遍扫描便可完成，因此比普通的中缀表达式的计算要简单得多。

例如，对于后缀表达式 $12\ 4\ -\ 5\ /\$ ，其中“ ”字符表示成分之间的空格，因减法运算符在前，除法运算符在后，所以应先做减法，后做除法；减法的两个操作数是它前面的 12 和 4，其中第一个数 12 是被减数，第二个数 4 是减数；除法的两个操作数是它前面的 12 减 4 的差(即 8)和 5，其中 8 是被除数，5 是除数。它对应的中缀表达式为： $(12-4)/5$ 。

这里的表达式包括：常量、变量和一般表达式。所以，前面看到的 $a=15$; $a=b$; $a=b+c$; 这些都是合法的赋值表达式。表达式也可以包括赋值表达式，例如：

```
a=b=10;
```

赋值表达式是一种特别的表达式，它的求值顺序是自右向左的。赋值表达式的运算过程为：先求解后面表达式的结果，然后将该结果赋给左侧的变量。赋值表达式不只是简单的赋值操作，它作为表达式的一种也有自己的值，这个值就是被赋值的变量的值，因此，表达式 $a=15$; 的值为 15。

而对于 $a=b=10$ ，该表达式的意义是先求解赋值表达式 $b=10$ 的值，结果为 10，然后将其赋给变量 a 。运算完成时， b 的值为 10， a 的值为 10，整个表达式 $a=b=10$ 的值也为 10。赋值运算符的结合性是从右至左的，因而 $a=b=10$ 与 $a=(b=10)$ 是等价的。

赋值运算的优先级相对于算术运算是较低的，因而当赋值表达式参与算术运算时，应在赋值表达式两端加上 $()$ ，否则编译器会报错。例如：

```
a=b+(c=4)
a=(b=5)/(c=4)
```

赋值表达式也可以包括复合赋值运算符，例如：

```
a+=b-=c*5;
```

设 a 的初值为 30， b 的初值为 30， c 的初值为 4，运算过程如下：先求 $c*5$ 的值，则结果为 20，然后求解 $b-=20$ ，结果为 10，最后进行 $a+=10$ 的运算，得最终结果 40。该表达式等价于 $a=a+(b=b-c*5)$ 。

4.6 关系运算符和关系表达式

C++ 中的关系运算符是用来对数据比较大小的，关系运算实际上是大小关系的比较。而且这里的大小关系与普通数学上的大小关系完全一致，如 $4>3$ 、 $a<7$ 等。关系运算的结果为真(true)或假(false)，当关系命题成立时取真，如 $4>3$ 的结果为 true；不成立时运算结果为假，如 $4<3$ 的结果为 false。六个关系运算符如表 4-2 所示。

表 4-2 关系运算符及其含义

关系运算符	含 义
>	大于
>=	大于等于
<	小于
<=	小于等于

续表

关系运算符	含 义
==	等于
!=	不等于

关系运算符的优先级低于算术运算符，高于赋值运算符。它们相互之间的优先次序为：>、<、>=、<=的优先级相同，==和!=的优先级相同，但比前面四种的要低。同级的运算符采用从左到右的顺序。

故关系式 $12 > 3 + 5$ 是合法的，先进行 $3 + 5$ 的运算，结果为 8， $12 > 8$ 成立，结果为真。但 $a > b = 4$ 是不合法的，因为关系运算符的优先级高于赋值运算符，先做 $a > b$ 的运算，结果为真(1)或假(0)，但 1 是数值不能向它赋值。正确的写法应为： $a > (b = 4)$ 。

用关系运算符将两个表达式连接起来的式子即为关系表达式。被连接的表达式可以是任意的表达式，下面的例子都是合法的关系表达式。

```
4 > 3;
X == ①Y
A >= b;
14 > 5 + 7;
5 > 3 > 2;
```

但是最后一个表达式 $5 > 3 > 2$ 初看起来结果应为真，但实际上并不是这样。由于两个 > 运算符相邻，按照从左到右的原则运算， $5 > 3$ 的结果为真(1)，再进行 $1 > 2$ 的运算得结果为假。要想正确的表达数学意义上的 $5 > 3 > 2$ ，写法应为 $5 > 3 \&\& 3 > 2$ ，表达式的意思是 $5 > 3$ 和 $3 > 2$ 都成立时该表达式取值为真，式中用到了与逻辑运算符“&&”，具体用法将在下一节看到。

① “==”用来表示等于运算符，而不是“=”，“=”号在C++中用来表示赋值操作，把这两者搞混是初学者常犯的错误。一般来说，这种错误编译器在进行语法检查时是很难发现的，是程序结果出错的常见根源，程序员在写程序时应尽量避免这种事情的发生。

“==”不可以用于对浮点型和双精度型的变量恒等的判断，这是因为，计算机内部是用尾数和阶码来表示浮点数的，这样就造成内存中记录的数据和真实值之间有很微小的误差(这个误差不影响计算精度)，因此在判断相等时，本来是相等的两个值，经过==计算的结果却是假。

C++中的假为 0，真是任意非 0 值。对于关系命题，成立时取系统给出的真值结果 1，但是当进行逻辑的真假判断时有可能通过其他非 0 值进行判断，在操作指针时，还会根据指针是否为空进行判断。这在实际工作中会经常遇到。

4.7 逻辑运算符和逻辑表达式

4.7.1 逻辑运算符

通常进行判断时,判断的条件有多种,这时需要将每一个判断表达为一个关系表达式,然后再看所有这些判断条件是否都达到了要求。逻辑运算符用来连接两个或多个关系表达式。逻辑运算符及其含义如表 4-3 所示。

表 4-3 逻辑运算及其含义

逻辑运算符	含 义
&&	与
	或
!	非

假设 A 和 B 是两个关系表达式,各逻辑操作的含义如下。

- 与(A&&B): 当且仅当 A 和 B 同时为真时,逻辑表达式 A&&B 的值才为真,否则为假。
- 或(A||B): 只有 A 和 B 中有一个为真或者 A 和 B 均为真, A||B 的值就为真,否则为假。
- 非(!A): 当 A 为真时, !A 的值为假;当 A 为假时, !A 的值为真。

逻辑运算同关系运算一样,运算结果为真时,用 1 来表示,结果为假时,用 0 来表示。逻辑表达式的值又叫做真值。对于某种形式的逻辑表达式,可以用一张真值表列出逻辑运算的各变量的真值和结果值。A、B 表示不同值时的真值表如表 4-4 所示。

表 4-4 A、B 表示不同值时的真值表

A	B	A&&B	A B	!A	!B
真	真	真	真	假	假
真	假	假	真	假	真
假	真	假	真	真	假
假	假	假	假	真	真

逻辑运算符之间的优先级关系是:逻辑非(!)高于逻辑与(&&), &&高于逻辑或(||)。例如:如果 a=10, b=3, c=5, 则!a||b&& c 的结果为真,该表达式等价于(!a)|| (b&& c); !a&& b||c

的结果也为真，该表达式等价于 $((!a)\&\&b)\|c$ 。

逻辑非(!)优先级高于算术运算符，逻辑与(&&)和逻辑或(\|)优先级低于关系运算符。

例如：

$(a>b)\&\&(c<d)$ 等价于 $a>b\&\&c<d$ 。

$(a>b)\|(c<d)$ 等价于 $a>b\|c<d$ 。

$(!a)\&\&(b<c)$ 等价于 $!a\&\&b<c$ 。

4.7.2 逻辑表达式

用逻辑运算符将若干个表达式连接起来即构成逻辑表达式。逻辑运算符不仅可以连接关系表达式，还可以连接常量、变量、算术表达式、赋值表达式甚至逻辑表达式本身。

逻辑表达式可以写得很长，但是如果一个表达式太复杂，则可以通过括号来保证运算次序，同时增强程序的可读性。

例如： $a<b==c==d$ 改成 $(a<b)==(c==d)$ 就清楚多了。又如： $x<10\&\&x+y!=20$ 最好改成 $(x<10)\&\&((x+y)!=20)$ 。

类似于关系表达式，逻辑表达式的计算结果为真时，系统自动将其值取 1，但在进行逻辑判断时，任意非 0 值作为逻辑运算符连接的一个表达式时都将被视为真，故： $a\|23$ 的结果为真。

逻辑表达式遵循从左到右的求值顺序，当前面的结果已经能确定该逻辑表达式的值时，将不再对后面的表达式内容求值，这被称作短路表达式。一般有下列两种典型的情况。

- **A&&B&&C 型**：设 $a=3$ ， $b=4$ ， $c=5$ ，表达式 $a\&\&(b-4)\&\&(c++)$ 的计算过程为： a 为 3 判断为真， $b-4$ 结果为假。由于是“与”操作，所以只要有一个为假，则整个表达式为假，后面的 $c++$ 将不再进行。因此程序进行完该句运算后， c 的值依然是 5，而不是 6。所以，想对变量进行赋值或有某些函数操作时，务必将表达式分成两步。
- **A\|B\|C 型**：这种类型的表达式当遇到 A、B、C 中有一个为真时，整个表达式即为真，后面的运算亦不再进行。

4.8 sizeof 运算符

`sizeof` 是单目运算符，它并不是函数，而是如同++、--等一样是运算符。`sizeof` 运算符以字节形式给出了其操作数的存储空间大小。操作数可以是一个常量、变量、表达式或是

括在括号内的类型名。操作数的存储大小由其类型决定。

`sizeof` 运算符返回变量或括号中的数据类型修饰符的字节长度。

用于数据类型的格式为：

`sizeof(数据类型修饰符)`

数据类型必须用括号括住，例如：

```
sizeof(double)
sizeof(int)
```

用于变量的形式为：

`sizeof(var_name)` 或 `sizeof var_name`

变量名可以不用括号括住。带括号的用法更普遍，大多数程序员采用这种形式。

但是 `sizeof` 运算符不能用于函数类型、不完全类型或位字段。不完全类型指具有未知存储大小的数据类型，如未知存储大小的数组类型、未知内容的结构或联合类型、`void` 类型等。如以下表达式都不是正确的形式。

```
sizeof(void)
sizeof(max)           //max 定义为 int max()
sizeof(char_v)        //char_v 定义为 char char_v [MAX] 且 MAX 未知
```

`sizeof` 运算符的结果类型是 `size_t`，它在头文件中 `typedef` 为 `unsigned int` 类型。该类型保证能容纳实现所建立的最大对象的字节大小。

`sizeof` 的优先级为 2 级，比 `/`、`%` 等 3 级运算符的优先级高。它可以与其他运算符一起组成表达式。例如：

```
i*sizeof(int)          //其中 i 为 int 类型变量
```

使用 `sizeof` 运算符的主要目的是为了增强程序的可移植性，使之不会由于不同机器在表示同一数据类型时存在字节长度不同而导致程序出错，从而确保程序可以在不同的硬件环境上正常运行。由于操作数的字节数在实现时可能出现变化，所以一般建议在涉及操作数字节大小时用 `sizeof` 来代替常量计算。

4.9 条件运算符和条件表达式

C++语言中有唯一的一个三目运算符——条件运算符“`?:`”，它要求有三个运算对象，可以把三个表达式连接构成一个条件表达式。条件表达式的一般形式如下：

表达式 1? 表达式 2 : 表达式 3

条件运算符的作用简单来说就是根据表达式 1 的值选择整个条件表达式的值。条件运算符的执行顺序为先求解表达式 1, 若为非 0(真)则求解表达式 2, 此时表达式 2 的值就作为整个条件表达式的值。若表达式 1 的值为 0(假), 则求解表达式 3, 表达式 3 的值就是整个条件表达式的值。要注意的是条件表达式中表达式 1 的类型可以与表达式 2 和表达式 3 的类型不一样。

下面是一个条件表达式的例子, 程序要求取出 a、b 两数中较小的值放入 min 变量中, 用条件运算符构成的条件表达式十分简单明了。

```
min = (a > b? a : b);
```

?: 运算符的第二个和第三个操作数决定了条件表达式的类型。设 A 和 B 为第二个和第三个操作数所属的类型。如果 A 和 B 的类型相同, 则此类型为该条件表达式的类型。否则, 两者中精度高的类型为条件表达式的类型。

条件运算符优先级高于赋值运算符, 低于关系运算符和算术运算符。条件运算符的结合方向为“自右向左”。如表达式:

```
a>b?a:c>d?c:d
```

相当于

```
a>b?a:(c>d?c:d)
```

如果 a=1, b=2, c=3, d=4, 则条件表达式的值等于 4。

很多时候, 条件运算符和后面讲到的 if-else 语句是可以互换的, 但是它的代码比后者少很多, 编译的效率也相对要高。但它也有着和复合赋值表达式一样的缺点, 可读性相对较差。在实际应用时要根据情况合理使用。

4.10 逗号运算符和逗号表达式

逗号运算符“,”的作用是连接两个表达式。它的优先级别在所有运算符中是最低的, 结合方向是“自左至右”的。

逗号表达式就是用逗号运算符将两个或几个表达式连接起来的式子。逗号表达式的一般形式是:

```
表达式 1, 表达式 2, 表达式 3, ..., 表达式 n;
```

逗号表达式的求解过程是: 先计算表达式 1 的值, 再计算表达式 2 的值, ……, 一直

计算到表达式 n 的值。最后整个逗号表达式的值是表达式 n 的值。例如：

```
x=8*2,x*4;           //整个表达式的值为 64，x 的值为 16
x=8*2,x*4,x*2;       //整个表达式的值为 128，x 的值为 16
x=(z=5,5*2);         //整个表达式为赋值表达式，它的值为 10，x 的值为 10，z 的值为 5
x=z=5,5*2;           //整个表达式为逗号表达式，它的值为 10，x 和 z 的值都为 5
```

逗号表达式用的地方不太多，一般情况是在给循环变量赋初值时才用得到。所以程序中并不是所有的逗号都要看成逗号运算符，尤其是在函数调用时，各个参数是用逗号隔开的，这时逗号就不是逗号运算符。

4.11 优先性和结合性

经过前面的学习，现将所有操作符的优先级总结一下，如表 4-5 所示，其中包括将在本书后面讨论的某些操作符。注意，所有操作符(除一元操作符和? 操作符之外)都是左结合的。一元操作符(*, &和-)及操作符“?”则为右结合。

在表中有些操作符出现了几次，请注意区分它们的区别。一般首先出现的是操作符的单目形式，第二次出现的是操作符的双目形式。

表 4-5 操作符优先级和结合性

优 先 级	操 作 符	结 合 性
1	() [] -> :: .	L→R
2	! ~ + - ++ -- & * (类型) sizeof new delete	R→L
3	. -> *	L→R
4	* / %	L→R
5	+ -	L→R
6	<< >>	L→R
7	< <= >= >	L→R
8	== !=	L→R
9	&	L→R
10	^	L→R
11		L→R
12	&&	L→R
13		L→R
14	? :	R→L

续表

优 先 级	操 作 符	结 合 性
15	= * = / = + = - = = << = >> =	R→L
16	,	L→R

本章小结

在本章中介绍了几种运算符和相应的表达式，并给出了左值和右值的概念，左值代表程序中使用的量所占用的空间，而右值代表实际的值。由表达式可以进一步组成语句，各种不同的运算符在进行运算时需遵循特定的优先级和结合性。

习 题

1. 已知 `int a=3; int b; int c`。执行下列运算后，`b` 和 `c` 的值分别是多少？

(1) `c=b=a+3`

`b` 和 `c` 的值分别是_____。

(2) `b=a++`

`c=++a`

`b` 和 `c` 的值分别是_____。

(3) `b=a+++a++`

`c=++a+++a`

`b` 和 `c` 的值分别是_____。

2. 什么是条件表达式，条件表达式的计算结果是什么类型？

数字解忧
PDG

第 5 章 程序的结构

本章内容：

- 顺序程序结构。
- 分支程序。
- 循环程序。

重点：

分支与循环程序的综合运用。

目的：

掌握程序设计的三种基本结构，掌握流程控制语句的使用方法和技巧。

从程序流程的角度来看，程序可以分为三种基本结构，即顺序结构、分支结构和循环结构。这三种基本结构可以组成所有各种复杂的程序。C 语言提供了多种语句来实现这些程序结构。C++ 语言脱胎于 C 语言，自然很好地继承了 C 在结构化程序设计上的优点，并加以发挥。本章将介绍这些基本语句及其在顺序结构中的应用，使读者对 C++ 程序有一个初步的认识，为后面各章的学习打下基础。

5.1 顺 序 结 构

顺序结构的程序从第一条语句到最后一条语句完全按顺序执行，其流程图如图 5-1 所示。

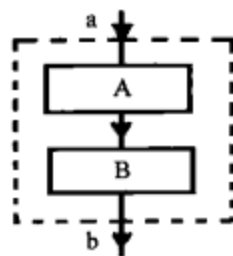


图 5-1 顺序结构流程示意

5.2 分支结构程序设计

分支结构的程序是指在程序的执行过程当中,根据用户的输入或中间结果去进行判断,以决定执行若干不同的任务。

分支程序的流程由多路分支组成,在程序的一次执行过程中,根据不同的情况,只有一条支路被选中执行,而其他分支上的语句被直接跳过,其流程图如图 5-2 所示。

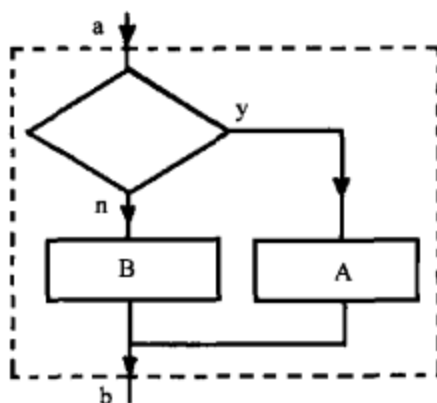


图 5-2 分支结构流程示意

分支程序包括三种结构:if...else...结构和 switch...case...break 结构,以及不常用的 goto 结构。

5.2.1 if...else...结构

if...else...结构分为以下三种。

1. 简单 if 语句

简单 if 语句的语法为:

```
if(表达式)
    语句;
```

或

```
if(表达式)
{
    语句块;
}
```

其中,if 后的表达式可以是在第 4 章讲过的任意表达式。当程序运行到 if 语句时,首

先计算表达式的值，当表达式为假(即取值为 0)时，条件不成立，则后面的语句或语句块不予执行，继续执行 if 结构之后的语句；当表达式为真(取任意非 0 值)时，条件成立，则执行后面紧跟着的语句或语句块。语句既可以是单条语句，也可以是由 {} 括起来的复合语句。

例如，在某玩家进入“城堡”时需要进行等级检查，假设超过 150 级的战士才能进入，其余的都将被挡在门外，其程序如下：

```
#include <iostream>
using namespace std;

void main()
{
    int playerlevel=180;
    if(playerlevel>150)
        cout<<"Player can get in";
}
```

2. if...else...语句

if...else...语句的语法为：

```
if(表达式)
    语句 1
else
    语句 2
```

这种结构使得程序可以在表达式的条件未成立时进入另一个处理过程，该结构非常适用于在两种情况之间作出判断的情况。

例如在上面的例子中，对于未达到等级而不允许进入的玩家，应当输出提示信息，其程序如下：

```
#include <iostream>
using namespace std;
void main()
{
    int playerlevel=180;
    if(playerlevel>150)
        cout<<"Player can get in";
    else
        cout<<"Player can't get in";
}
```

3. if...else if...语句

在实际应用中，选择常常面对更多的分支，这时简单的 if...else 语句就不能满足需要

了。将 if...else 语句扩展一下，就可以得到 if...else if...结构，其一般形式为：

```
if <表达式 1 >  
    语句 1  
else if <表达式 2 >  
    语句 2  
else if <表达式 3 >  
    语句 3  
else 语句 4
```

有了 if...else if...语句后，就可以对多项条件进行选择，程序执行的流程图如图 5-3 所示。

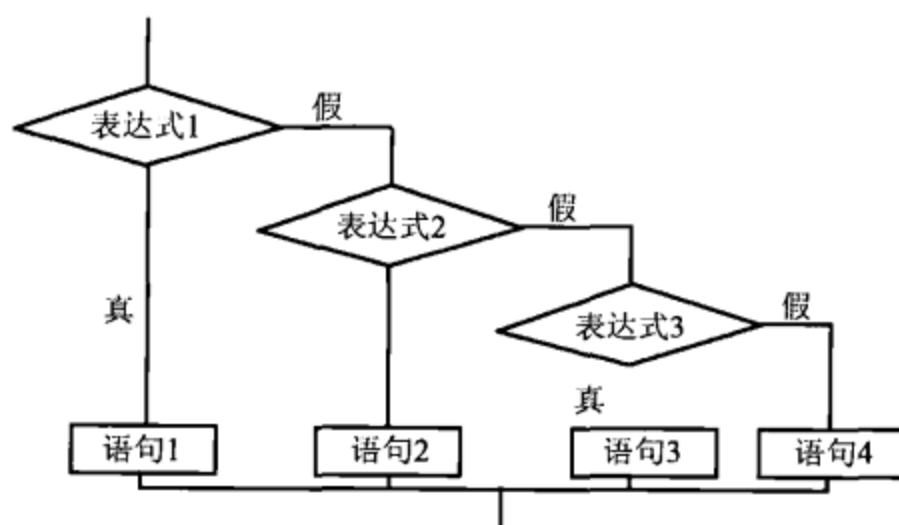


图 5-3 if...else if...语句流程图

例如，战士在受到攻击时，由于自身身体状况的好坏，躲避开攻击的概率导致被打中时受伤的轻重程度都有明显不同。在实际编写程序时，通常是先将身体状况分级，然后根据不同的级别做相应的处理。

假设将一个运动员的身体状态分为 4 级：好、一般、较差和很差。状态好时几乎不受到伤害，一般被击中的可能性设为 20%，较差时 50%，很差时为 80%。则在程序中判断受伤时可按如下例子处理：

```
#include <iostream>  
using namespace std;  
  
int getposib(float posib)  
{  
    ...  
}  
  
void main()  
{
```

```

int Status = 1;
int life = 300;
int Hurt = 0;
int attack = 30;

if(status ==1)      //好
{
    Hurt += attack*getposib(0.05);
}
else if(status ==2) //一般
{
    Hurt += attack*getposib(0.2);
}
else if(status ==3) //较差
{
    Hurt += attack*getposib(0.5);
}
else if(status ==4) //很差
{
    Hurt += attack*getposib(0.8);
}
Life -= hurt;
cout<<"生命剩余值"<<Life;
}

```

以上程序中的 `int getposib(float posib)` 是一个根据概率计算是否受伤的函数，参数为受伤概率，若受伤返回 1，否则为 0。这样，上述程序便完成了根据身体状态进行受伤判断的计算。

在使用 `if...else...` 结构时应注意避免程序的二义性。如果把 `if...else...` 语句和 `if` 语句嵌套结合可得到以下结构。

```

if(表达式 1)
    if(表达式 2)
        语句 1
    else
        语句 2

```

其中的 `else` 是在表达式 1 不成立时执行，还是在表达式 2 不成立时执行呢？C++ 中规定：`else` 与它最临近的 `if` 语句相配合。则上述表达式等价于：

```

if(表达式 1)
{
    if(表达式 2)
    {
        语句 1
    }
}

```



```

    }
    else
    {
        语句 2
    }
}

```

为了使 if...else...结构在使用时更为清晰,建议在书写时如同上面的程序段一样,将成对的大括号与语句块对齐。这样,程序可读性较好,结构清晰,一目了然。

5.2.2 switch 语句

switch 语句是专门为多路分支设计的。其语法格式为:

```

switch(表达式)
{
    case 常量表达式 1: 语句 1
    case 常量表达式 2: 语句 2
    :
    case 常量表达式 n: 语句 n
    default: 语句 n+1
}

```

它表达的意义是:当“表达式”的值等于“常量表达式 1”时,从“语句 1”开始执行到结构结束;当“表达式”的值等于“常量表达式 2”时,从“语句 2”开始执行;当“表达式”的值不同于所有 n 个常量表达式的值时,从“语句 n+1”开始执行。图 5-4 表示了 switch 语句的流程。

需要注意,表达式的值与某一常量表达式匹配后,执行相应的语句,但执行完之后并不自动跳出 switch 语句。默认情况下,程序将一次执行完该 case 条件以后的各个条件中要执行的语句。

若要在执行完一个 case 对应的语句后即跳出 switch 语句,应在相应的语句后添加 break 语句。修改后的 switch 语句格式如下:

```

switch(表达式)
{
    case 常量表达式 1: 语句 1
                        break;

```

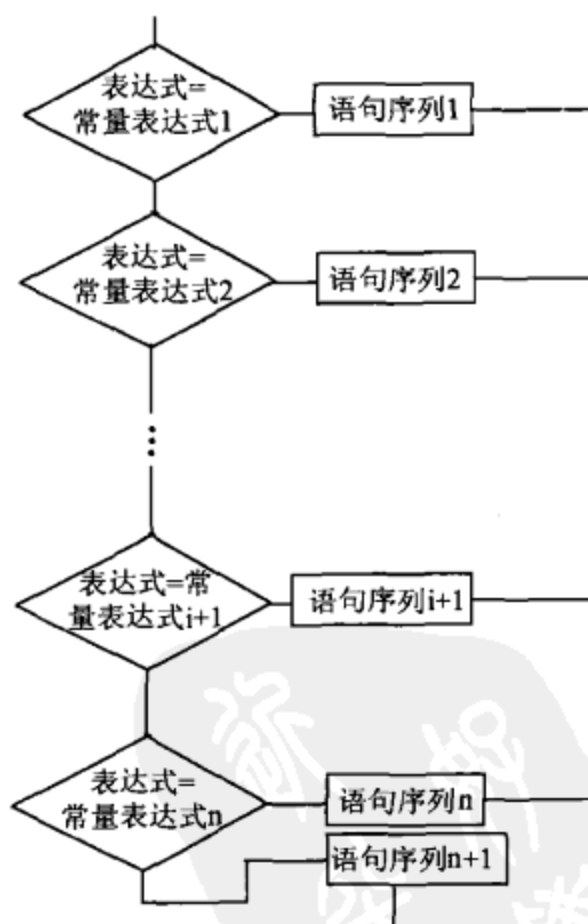


图 5-4 switch 语句的流程图

```

case 常量表达式 2: 语句 2
                    break;
:
case 常量表达式 n: 语句 n
                    break;
default: 语句 n+1
}

```

break 语句的功能是终止对 **switch** 语句或循环语句的执行，即跳出这两种语句，而转入下一语句执行，加 **break** 语句后的流程图如图 5-5 所示。

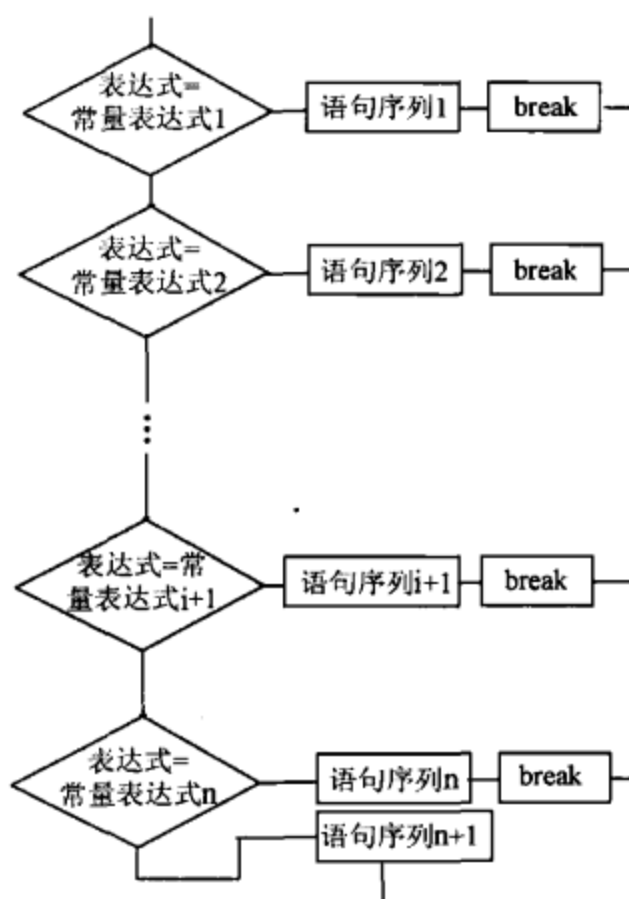


图 5-5 加 **break** 语句后的流程图

例如：想要根据考试成绩的等级(grade)打印出百分制分数段，可以通过下面的 **switch** 结构来选择合适的分数段：

```

switch (grade)
{
    case 'A': cout<<"85~100"<<endl;
    case 'B': cout<<"70~84"<<endl;
    case 'C': cout<<"60~69"<<endl;
    case 'D': cout<<"<60"<<endl;
    default: cout<<"error"<<endl;
}

```

但是，如果这样写程序，当 **grade='A'** 时，程序从 **cout<<"85~100"<<endl** 语句开始执行，

将会不停止地执行下面所有代码，因此输出结果为：

```
85~100
70~84
60~69
<60
error
```

本例中，原意是输出一个值，要实现这一要求，应在语句后加 `break` 语句。改正后的程序如下。

```
switch (grade)
{
    case 'A': cout<<"85~100"<<endl;
              break;
    case 'B': cout<<"70~84"<<endl;
              break;
    case 'C': cout<<"60~69"<<endl;
              break;
    case 'D': cout<<"60"<<endl;
              break;
    default: cout<<"error"<<endl;
}
```

`break` 语句使程序执行一次就跳出 `switch` 结构，加入 `break` 语句后使本例达到设计的要求。



注意：① `if...else...` 语句多用于两路分支，虽然第三种情况使用 `if...else if...` 语句或用 `if...else...` 的嵌套结构也可以实现多路分支，但当分支的情况比较多时，这样处理不仅增加了程序代码的长度，而且使程序条理性变差，降低了可读性。`switch` 语句专为多路分支设计，建议在进行多路处理时多使用这种结构。

② 跳出 `switch` 语句应使用 `break` 语句，忘记加 `break` 语句是很多程序出错的原因。

③ 出于程序的完整性和健壮性的考虑，建议对每个 `switch` 语句结构都不要省略其中的 `default` 语句，即使什么也不做，最好也列一条空语句在那里，因为并不能保证人的分析是绝对全面的。

④ `switch` 语句后括号中的变量或表达式的值必须为有序类型^①。

① 有序类型是指：在变量类型的范围内，除了第一个数据之外，其他所有的数据，有且仅有一个直接前趋，除了最后一个数据之外，其他数据有且仅有一个直接后继，例如整型就是有序类型。

5.2.3 goto 语句

goto 语句^①是一种无条件转移语句，与 Basic 中的 goto 语句相似。goto 语句的使用格式为：

```
goto 标号;
```

其中标号的定义应符合 C++ 中标识符的定义规则，当这个标识符加上一个 “:” 一起出现在程序内某处的，执行 goto 语句后，程序将跳转到该标号处并执行其后的语句。

另外标号必须与 goto 语句同处于一个函数中，但可以不在一个循环层中。通常 goto 语句与 if 条件语句连用，当满足某一条件时，程序跳到标号处运行。例如：

```
#include <iostream>
void main()
{
    int i=0;
    char c;
    cout<<"\ Esc----Quit\ "<<endl;
    cout<<"\ Enter----Next line\ "<<endl;
    while(1)
    {
        c='\0';
        while(c!=13)
        {
            c=getch();
            if(c==27)
                goto quit;
            cout<<c;
        }
        cout<<endl;
        i++;
        cout<<"The line is "<<i<<endl;
    }
    quit:cout<<"The end"<<endl;
}
```

当程序执行到 “goto” 时，将直接跳转到最后一行执行。

goto 语句的一个重要应用是与 if 语句连用构成循环，这种使用在早期的 Basic 等语言中非常普遍。

① goto 语句违反了结构化设计的原则，不推荐使用。

例如，用 if 语句和 goto 语句构成循环，求 $\sum_{n=1}^{100} n$ 的值。其代码如下：

```
#include <iostream>
using namespace std;
void main()
{
    int i, sum=0;
    i=1;
loop:
    if(i<=100)
    {
        sum=sum+i;
        i++;
        goto loop;
    }
    cout<<"sum="<<sum<<endl;
}
```

使用 goto 语句执行的跳转非常灵活，可以跳到程序的任何位置。但是也正因如此，goto 语句违反了程序的结构化原则，使程序层次不清，且大大降低了可读性。通常情况，不推荐使用 goto 语句。只有在极少数多层嵌套退出或其他非常复杂的跳转场合，使用 goto 语句可以大大提高效率时才少量使用。

5.3 循环结构程序设计

如果在程序的某处，需要根据某项条件重复地执行某项任务若干次或直到不满足某条件为止，就构成了循环结构，其流程图如图 5-6 所示。

循环结构可以使程序只包含很少的语句，而让计算机反复执行，从而完成大量同类的计算。构成循环的方式有三种：for 语句、while 语句和 do-while 语句。

5.3.1 for 语句

for 语句常用于循环次数已知的循环控制，或是与计数相类似的问题，也可以灵活地用于其他循环控制。它的一般形式为：

```
for(表达式 1;表达式 2;表达式 3)
```

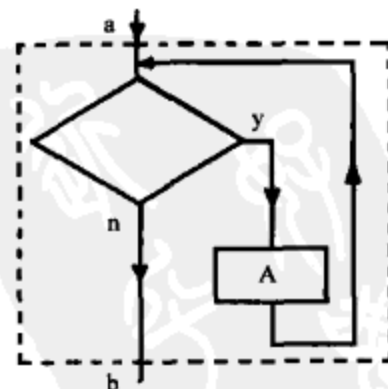


图 5-6 循环结构示意图

```

{
    循环体
}

```

for 语句通过一个循环控制变量来控制循环体部分的循环过程。在每一次循环执行前检查循环变量是否仍满足循环条件，是则进行一次循环，且改变一次循环变量；否则结束循环。其中，表达式 1 用作循环控制变量的初始化语句，用来给循环控制变量赋初值；表达式 2 称条件表达式，是一个关系表达式，它决定什么时候退出循环；表达式 3 定义循环控制变量每循环一次后按什么方式变化，这三个部分之间用“;”分开。

for 语句的执行过程如下。

- ① 初始化控制变量。
- ② 求解条件表达式②，若为“真”，则执行“语句”；若为假，转第⑤步。
- ③ 执行表达式 3，改变循环变量的值。
- ④ 转第②步。
- ⑤ 结束循环，执行 for 语句后面的语句。

图 5-7 所示为 for 循环执行的流程。

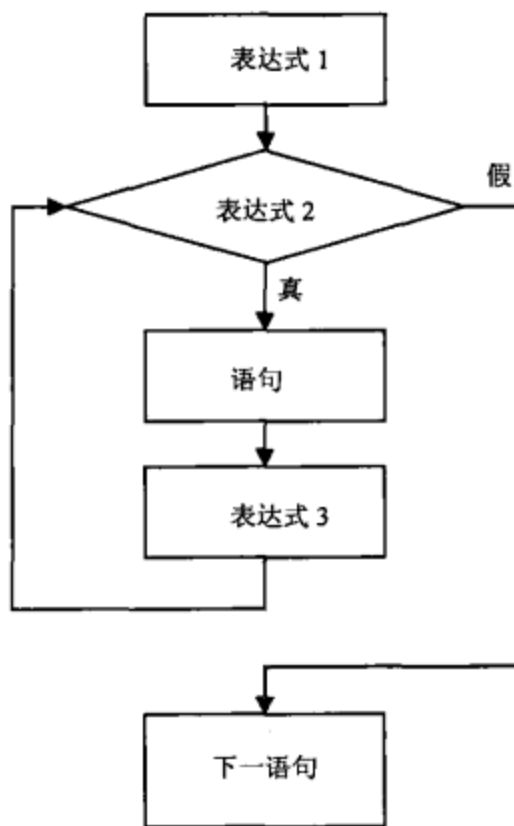


图 5-7 for 语句的执行流程

很多实际问题在程序的结构设计上都可以使用循环结构，特别是有着一定规律的大量重复计算。

例如，计算从 1 加到 10 的结果，其程序如下。

```
#include <iostream>
using namespace std;
void main()
{
    int i=0,sum=0,k=0;
    for(i=1; i<=10; i++)
    {
        k=k+1;
        sum = sum + k;
    }
    cout<<"sum="<<sum<<endl;
}
```

上例中先给循环变量 *i* 赋初值 1，判断 *i* 是否小于等于 10，若是则执行语句“sum=sum+i;”，之后 *i* 值增加 1。再重新判断，直到条件为假，即 *i*>10 时结束循环。

仔细观察，就会发现：*k* 的值和 *i* 的值是始终保持一致的。因此可以将上面的程序改进如下。

```
#include <iostream>
using namespace std;
void main()
{
    int i,sum=0;
    for(i=1; i<=10; i++)
    {
        sum = sum + i;
    }
    cout<<"sum= " <<sum<<endl;
}
```

这里的 *i* 不仅作为循环变量，而且也巧妙地被用作每一次的加数，正好在十次循环后得到结果。

for 语句的循环变量初始化表达式还可以同时是循环变量的定义式。也就是说，在要使用到循环变量时再来定义它。C++支持在任何位置定义变量，包括在 for 语句中。上面的例子可改成：

```
int sum=0;
for (int i=1; i<=10; i++)
```

这样定义的循环变量就被限制在定义它的循环的域范围内(域的概念在后面会讲到)，仅为这个循环所使用，一般都这样定义循环变量。

实际上, for 的表达式 1、2、3 可以为任何表达式, 而不仅限于以上的形式。例如: 表达式 1 可以是逗号表达式:

```
int i, sum;
for(i=1, sum=0; i<=10; i++)           //同时进行循环变量和结果变量的初始化
{
    sum = sum + i;
}
```

表达式 2、3 也可以是逗号表达式。利用这一特性, 甚至不用循环体, 在 for 语句中即可完成循环运算。例如, 对于求和运算, 还可以写成这样:

```
for(i=1, sum=0; i<=10; sum+=i++)
{
    //循环体为空
}
```

这样的写法在某些情况下是有用的, 但可读性不高, 因此并不提倡频繁地使用。

在 for 循环语句 for(初始化表达式;条件表达式;增量表达式)中, “初始化表达式”、“条件表达式”和“增量表达式”都是可选项, 即它们是可以省略的, 但“;”不能省略。省略不同的表达式有不同的效果。

(1) 省略表达式 1。表达式 1 的作用是设定循环初始条件, 表达式 1 省略后, 应在 for 语句之前设置循环初始条件。例如:

```
for(;i<=100;i++)                     //注意, 表达式 1 后面的分号不能省略
{
    sum = sum + i;
}
```

(2) 省略表达式 2。表达式 2 为循环条件, 省略后即不判别循环条件, 认为循环条件始终为“真”, 循环将无终止地进行下去。例如:

```
for(i=1;;i++)
{
    sum = sum + i;
    if(i==10)
        break;
}
```

为了防止程序进入死循环, 所以在这样的循环体里一定要用别的手段来结束循环, 如使用 break 语句。

(3) 省略表达式 3。省略了修改循环条件的语句, 同样的, 程序应在循环体中修改循环条件, 以保证循环能正常结束。例如:

```
for(sum=0,i=1;i<=100;)
{
    sum = sum + i;
    i++;
}
```

(4) 全部省略。这时的 for 语句形式为：

```
for( ; ; )
{
    ...
}
```

这是一种简单的死循环形式。

5.3.2 while 语句

while 语句是另一种循环控制结构语句，通过不断检查一个循环条件，在满足条件的情况下循环，不满足即退出循环结构。while 循环的一般形式为：

```
while(条件表达式)
{
    语句;
}
```

while 循环表示当条件表达式为真时，便执行循环体语句，直到条件为假时才结束循环，并继续执行循环的后续语句其流程图如图 5-8 所示。

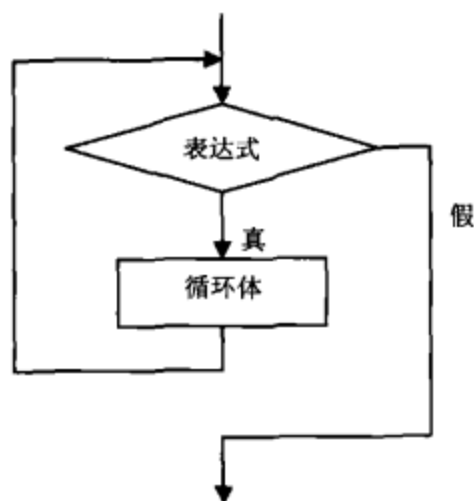


图 5-8 while 循环流程图

以下示例用于求 10 的阶乘(10)!。

分析： $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$ ， $0! = 1$ 。即 $S_0 = 1$ ， $S_n = S_{n-1} * n$ 。可以从 S_0 开始，依次求出 S_1 、 S_2 、 \cdots 、 S_n 。令 S 为最终结果阶乘值， S 的初值为 $0! = 1$ ；以变量 i 为计数器， i 从 1

变到 n ，每一步令 $S=S*i$ ，则最终 S 中的值就是 $n!$ 。程序如下：

```
#include <iostream>
using namespace std;
void main()
{
    int n,i;
    long int s;    //请思考为什么要使用长整型?
    s=1;
    i=1;
    while (i<=10)
    {
        s*=i;
        i=i+1;
    }
    cout<<"10!="<<s<<endl;
}
```

程序运行结果如下：

10!=3628800

程序流程图如图 5-9 所示。

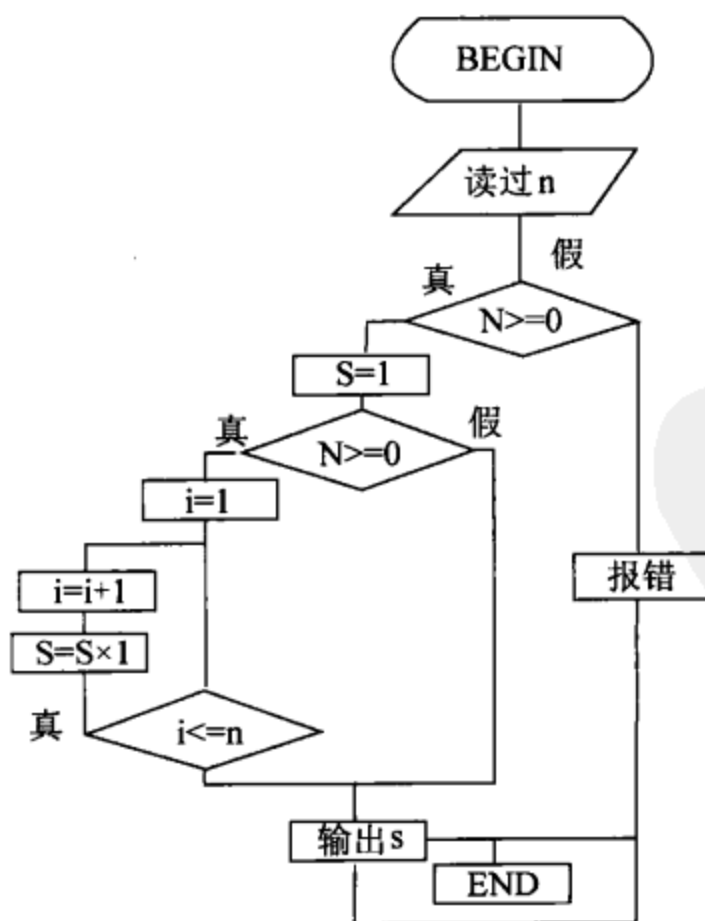


图 5-9 程序流程图

5.3.3 do-while 语句

do-while 循环的一般格式为：

```
do
{
    循环体
}
while(条件);
```



注意： 要注意 do-while 语句的结尾有分号，而 while 语句则没有。

do-while 循环与 while 循环的不同在于：while 语句是先判断，后执行，若在第一次进入循环时条件就不成立，则循环体一次也不执行；而 do-while 循环先执行循环中的语句，然后再判断条件是否为真，如果为真则继续循环，如果为假则终止循环。因此，do-while 循环至少要执行一次循环语句。这是 while 和 do-while 循环的最大不同。do-while 循环流程图如图 5-10 所示。

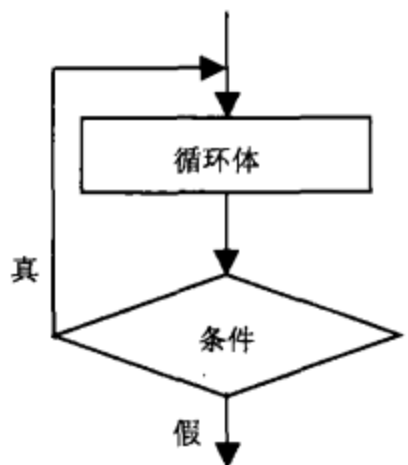


图 5-10 do-while 循环流程图

以下示例用 do-while 语句求 $\sum_{n=1}^{100} n$ 的值。

分析： $\sum_{n=1}^{100} n = 1+2+3+\cdots+100$ 。即 $S_0=1$ ， $S_n=n+S_{n-1}$ 。可以从 S 开始，依次求出 S_1 、 S_2 、 \cdots

S_n 。令 sum 保存求和结果，设 sum 的初值为 0；变量 i 为计数器，i 从 1 变到 n，每一步令 sum 加 i，则最终 sum 中的值就是 $\sum_{n=1}^{100} n$ 的值。程序如下：

```
#include <iostream>
using namespace std;
void main()
```

```

{
    int i, sum=0;
    i=1;
    do
    {
        sum+=i;
        i++;
    }
    while(i<100);
    cout<<"sum 1~100="<<sum<<endl;
}

```

程序流程图如图 5-11 所示。

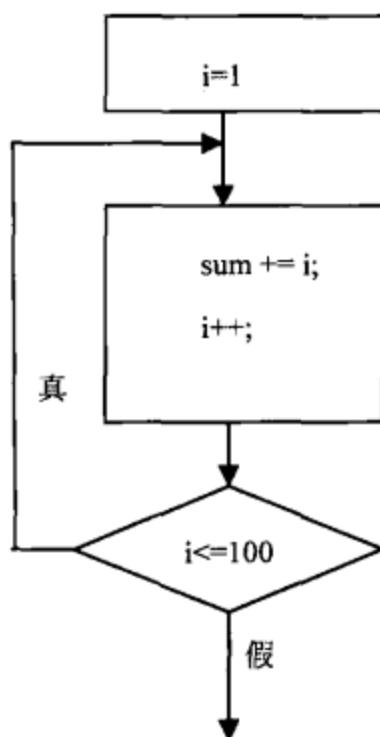


图 5-11 程序流程图

5.3.4 循环的嵌套

循环嵌套是指一个循环(称为“外循环”)的循环体内包含另一个循环(称为“内循环”)。内循环中还可以包含循环,形成多层循环。理论上,循环嵌套的层数可以是无限层。

三种循环(while 循环、do-while 循环和 for 循环)可以互相嵌套。例如:

(1) while 循环嵌套 while 循环。

```

while()
{
    while()
    {

```

```
        ...  
    }  
}
```

(2) for 循环嵌套 while 循环。

```
for(...;...;...)  
{  
    while()  
    {  
        ...  
    }  
}
```

(3) do-while 循环嵌套 for 循环。

```
do  
{  
    ...;  
    for(...;...;...)  
    {  
        ...;  
    }  
    ...  
}  
while();
```

除了上面列出的，其他循环结构语句也可以互相嵌套，这里就不一一列出了。多重循环的使用与单一循环完全相同，但应特别注意内、外层循环条件的变化，以便正确控制循环次数。

5.4 break、continue 语句

5.4.1 break 语句

前面用到了 **break** 语句，**break** 语句通常用在 **switch** 语句和循环语句中。当 **break** 语句用于选择语句 **switch** 中时，可使程序跳出 **switch** 而执行 **switch** 以后的语句，**break** 语句在 **switch** 中的用法已在前面介绍，这里不再举例。

当 **break** 语句用于 **do-while**、**for** 和 **while** 循环语句中时，可使程序终止循环而执行循环后面的语句。通常 **break** 语句总是与 **if** 语句连在一起，以在满足或不满足条件时跳出循

环，示例如下：

```
#include <iostream>
using namespace std;
void main()
{
    int i=0;
    char c;
    cout<<"\ Esc----Quit\ "<<endl;
    cout<<"\ Enter----Next line\ "<<endl;
    while(1)                //设置为无限循环
    {
        c='\0';              //变量赋初值
        while(c!=13&&c!=27)    //键盘接收字符直到按 Enter 键或 Esc 键
        {
            c=getch();        //获取键盘输入字符
            cout<<c;
        }
        cout<<endl;
        if(c==27)
            break;            //判断若按 Esc 键则退出循环
        i++;
        cout<<"The line is"<<i<<endl;
    }
    cout<<"The end"<<endl;
}
```

该程序为统计用户输入字符行数的程序，按 Enter 键表示一行数据输入完毕，按 Esc 键表示所有输入完毕，程序中通过判断输入字符的值来确定是不是 Esc 键，若是则执行 break 语句跳出循环，结束程序。



注意： break 语句只用在 switch 语句和循环语句结构体中；仅在 if...else 的条件语句中使用 break 语句，或在任何 switch 和循环结构以外的位置使用都会导致编译错误。

例如：

```
void main()
{
    ...
    if()
    {
        ...
        break;        //编译出错
    }
    ...
}
```



```

    }

```

在类似上面的程序中，**break** 语句的使用就是错误的。但如果将 **if...else** 语句放在循环体内就没有问题了，例如：

```

void main()
{
    ...
    for( ; ; )
        if()                //if 语句是循环体
        {
            ...;
            break;           //跳出的是 for 循环
        }
    ...
}

```

这里的 **break** 语句表示的是跳出 **for** 循环，而不是 **if** 结构。

如果在多层循环中，一个 **break** 语句只向外跳出一层循环，即结束 **break** 语句当前所在循环的进行，而对外层循环没有影响。

5.4.2 continue 语句

continue 语句的作用是跳过循环体中剩余的语句而强行执行下一次循环。**continue** 语句只用在 **for**、**while** 和 **do-while** 等循环体中，常与 **if** 条件语句一起使用，用来加速循环。

continue 对于 **while** 和 **do-while** 循环，意味着立即求解循环是否终止的表达式(即执行条件测试部分)；而对于 **for** 语句来讲，这意味着立即求解表达式 3。

例如：从键盘上输入 10 个字符，并统计其中数字字符的个数。其程序如下：

```

#include <conio.h>
void main()
{
    int sum,i;
    char ch;
    sum=0;
    for(i=0;i<10;i++)
    {
        ch=getchar();
        if(ch<'0' || ch<'9')
            continue;
        sum++;
    }
    cout<<"sum="<<sum<<endl;
}

```



在本程序中，当读入的字符(在 ch 中)不是数字字符('0'~'9')时，将不执行 sum++ 语句，而是立即执行下一轮循环。

本章小结

本章阐述了 C++ 语言面向过程的三种基本结构：顺序结构、分支结构和循环结构。分支结构有 if...else 语句和 switch 语句两种。循环结构包括 for 语句、while 语句和 do-while 语句。各种语句之间还可以相互嵌套使用。另外还介绍了 break、continue 和 goto 语句。

习 题

1. 假设本题中所有的问题全处于理想情况：已知一只新出生的母牛，到第 4 年可以成熟并会新生一只小牛，又新生的小牛一定是母牛，在这样的情况下，试求第 N 年一共有多少头牛。N 从键盘上读入。

2. 已知水仙花数的定义是：一个三位的十进制数，每个数据位上的数字的立方和如果等于该数，则该数为水仙花数。请用两种不同的设计思路求出从 100~999 之间的所有水仙花数。

3. 已知一种记分方式为 100 分制，另外一种记分方式为 A~E，两种记分方式之间数据的转换原则为：A 对应 90~100，B 对应 80~89……E 对应 0~59。试写程序，将 100 分制的分值转换为字符记分的分值。



第 6 章 宏和编译预处理

本章内容:

- 宏的定义及使用。
- 编译预处理。
- 条件编译。
- 更多的编译预处理指令。

重点:

- 条件编译。
- 编译预处理指令。

目的:

掌握编译预处理技巧, 协调与控制大型工程的编译过程。

编译预处理: 在源程序文件中, 加入“编译预处理命令”, 使编译程序在对源程序进行通常的编译(包括词法分析、语法分析、代码生成和代码优化)之前, 先对这些命令进行预处理, 然后将预处理的结果和源程序一起再进行通常的编译处理, 以得到目标代码(obj 文件)。

C++提供的编译预处理命令包括三种: 宏命令(macro)、文件包含命令(include)和条件编译命令。

ANSI 的标准规定, 预处理指令主要包括以下几个。

```
#define
#error
#if
#else
#elif
#endif
#ifdef
#ifndef
#undef
#line
#pragma
```



由上述指令可以看出, 每个预处理指令均带有符号“#”。此外, 每条预处理指令必须独占一行。例如`#include<stdio.h> #include <iostream>`的书写形式是错误的。

下面介绍一些常用指令。

6.1 宏 定 义

`#define` 指令是一个宏定义指令, 定义的一般形式是:

`#define` 宏替换名字符串(或数值)

由`#define` 指令定义后, 在程序中每次遇到该宏替换名时就用所定义的字符串(或数值)代替它。

例如: 可用下面语句定义 `TRUE` 表示数值 1, `FALSE` 表示 0。

```
#define TRUE 1
#define FALSE 0
```

一旦在源程序中使用了 `TRUE` 和 `FALSE`, 编译时会自动地用 1 和 0 代替。



- 注意**
- ① 在宏定义语名后没有“;”。
 - ② 习惯上用大写字符作为宏替换名, 而且常放在程序开头。
 - ③ 宏定义还有一个特点, 就是宏替换名可以带有形式参数, 在程序中用到时, 实际参数会代替这些形式参数。

例如:

```
#define MAX(x, y) (x>y)?x:y
main()
{
    int i=3, j=4;
    printf("两数中较大者为 %d", MAX(i, j));
}
```

上例中宏定义语句的含义是用宏替换名 `MAX(x, y)` 代替 `x, y` 中的较大者, 同样也可进行如下定义:

```
#define MIN(x, y) (x<y)?x:y
```

以下几种情况是宏的高级应用。

1. 不带参数的宏

定义不带参数的宏的目的是为了用一个指定的名称代替一个常量或是一个字符串。定义的一般形式为：

```
#define 标识符 字符串
```

如：`#define PI 3.1415926`的作用是用标识符(称为“宏名”)PI代替字符串“3.1415926”。在预编译时，编译程序会将源程序中出现的宏名PI替换为字符串“3.1415926”，这一替换过程称为“宏展开”。

`#define`是宏定义命令；`#undef`是终止宏定义命令。下面是一个宏定义及其应用的例子，程序如下：

```
#include <iostream>
using namespace std;
#define PI 3.1415926

void main()
{
    float l,s,r,v;
    cout<<"请输入圆的半径:";
    cin>>r;                /* 输入圆的半径
    l = 2.0f*PI*r;          /* 圆周长
    s = PI*r*r;             /* 圆面积
    v = 4.0f/3.0f*PI*r*r*r; /* 球体积
    cout<<"l="<<l<<endl<<"s="<<s<<endl<<"v="<<v<<endl;
}
```

程序运行结果如下：

请输入圆的半径:4(用户输入)

```
l=25.1327
s=50.2655
v=268.083
```

宏定义中可以引用已定义的宏名。例如：

```
#include <iostream>
using namespace std;

#define R 3.0
#define PI 3.1415926
#define L 2*PI*R
#define S PI*R*R
```



```
void main()
{
    cout<<"L="<<(L)<<endl<<"S="<<(S)<<endl;
}
```

程序运行结果如下：

```
L=18.8496
S=28.2743
```

关于宏定义的几点说明如下。

- (1) 一般宏名用大写字母表示(变量名一般用小写字母)。
- (2) 使用宏可以提高程序的可读性和可移植性。如在上述程序中，多处需要使用 π 值，用宏名既便于修改又意义明确。
- (3) 宏定义是用宏名代替字符串，宏扩展时仅作简单替换，不检查语法。语法检查在编译时进行。
- (4) 宏定义不是 C++ 语句，后面不能有分号。如果加入分号，则连分号一起替换。
例如：

```
#define PI 3.1415926;
area = P*r*r;
```

在宏扩展后成为：

```
area = 3.1315926; *r*r;
```

结果，在编译时就会出现语法错误。

- (5) 通常把 `#define` 命令放在一个文件的开头，使其在本文件全部有效(`#define` 定义的宏仅在本文件有效，在其他文件中无效，这与全局变量不同)。
- (6) 对程序中用双引号括起来的字符串，即使与宏名相同，也不替换。例如：

```
#include <iostream>
using namespace std;
#define STR "Welcome to C++"
void main()
{
    cout<<"STR"<<endl;
    cout<<STR<<endl;
}
```

本例中的第一个输出语句中由于 `STR` 用 “” 括起来，表明这是一个普通的字符串，不是宏，编译时不被替换。而下一个输出语句中 `STR` 才代表定义的宏，在编译时将被替换为 “Welcome to C++”。

程序运行结果如下:

```
STR
Welcome to C++
```

2. 带参数的宏

`#define` 指令的另一个重要功能是定义带参数的宏。每当编译器遇到宏名字时, 与之有关的参数都由程序中的实际参数替换。由于看上去和函数类似, 因此这一宏形式被称为类函数宏。定义的一般形式为:

`#define 宏名(参数表) 字符串`

带参数的宏在展开时, 不是进行简单的字符串替换, 而是进行参数替换, 如图 6-1 所示。



图 6-1 带参数宏展开示例

示例如下:

```
#include <iostream>
using namespace std;
#define PI 3.1415926
#define S(r) PI*r*r

void main()
{
    float a, area;
    a = 3.6;
    area = S(a);
    cout<<"r="<<a<<endl<<"area="<<area<<endl;
}
```

程序运行结果如下:

```
r=3.6
area=40.715
```

说明:

① 带参数的宏展开时, 用实参字符串替换形参字符串, 要注意可能发生的错误。比较好的办法是宏定义的形参加括号, 如表 6-1 所示。

② 宏定义时, 宏名与参数表间不能有空格。

例如: `#define S(r) PI*r*r` (表示空格)是错误的

带参数的宏定义与函数的区别如表 6-2 所示。

表 6-1 几种宏的常用方法展开后的表达式

宏 定 义	语 句	展 开 后	是否正确
#define S(r) PI*r*r	area = S(a+b);	area = PI*a+b*a+b;	错误
#define S(r) PI*(r)*(r)	area = S(a+b);	area = PI*(a+b)*(a+b)	正确

表 6-2 带参数的宏与函数的区别

比较项目	函 数	宏
信息传递	实参的值或地址传送给形参	用实参的字符串替换形参
处理时刻及内存分配	程序运行时处理，分配临时内存单元	宏展开在预编译时处理，不存在分配内存的问题
参数类型	实参和形参类型一致。如不一致，编译器将进行类型转换	字符串替换，不存在参数类型问题
返回值	可以有一个返回值	可以有多个返回值
对源程序的影响	无影响	宏展开后使程序加长
时间占用	占用程序运行时间	占用编译时间

利用宏可以完成一个表达式返回多个结果，例如：

```
#include <iostream>
using namespace std;

#define PI 3.1415926
#define CIRCLE(R,L,S,V) L=2*PI*R;S=PI*R*R;V=4/3*PI*R*R*R

main()
{
    float r,l,s,v;           //半径、圆周长、圆面积、球体积
    cout<<"Input the Radius: ";
    cin>>r;
    CIRCLE(r,l,s,v);
    cout<<"r="<<r<<" , l="<<l<<" , s="<<s<<" , v="<<v<<endl;
}
```

程序运行结果如下：

```
Input the Radius: 23
r=23,l=144.513,s=1661.9,v=38223.8
```

3. 宏定义终止命令

`#undef` 指令的功能是结束先前定义的宏名。定义的一般格式为：

`#undef` 宏名

例如：

```
#define G 9.8
main()
{
}
#undef G      /* 取消 G 的意义 */
f1()
:
:
```

`#undef` 指令主要用于使宏名局部化，使其仅局限于使用该宏的代码段。

6.2 头文件包含

程序中的`#include`为文件包含命令。该命令指定编译程序在预处理时，把另一指定文件的内容复制到本文件，再对合并后的文件进行编译。该命令的一般格式是：

`#include "文件名"`

或

`#include <文件名>`

一个程序包含两个文件 `file1.c` 和 `file2.c`，在 `file1.c`，在 `file1.c` 中有(`#include "file2.c"`)字样，如图 6-2 所示。

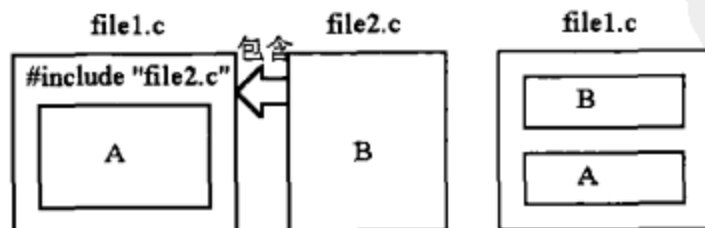


图 6-2 文件包含示意

在 `file1.c` 文件中，有文件包含命令`#include "file2.c"`，预处理时，先把 `file2.c` 的内容复制到文件 `file1.c` 中，再对 `file1.c` 进行编译。

从理论上说，`#include` 命令可以包含任何类型的文件，只要这些文件的内容被扩展后符合 C++ 语法。一般 `#include` 命令用于包含扩展名为 `.h` 的“头文件”，如 `stdio.h`、`string.h`

和 `math.h`。在这些文件中，一般定义符号常量、宏或声明函数原型。程序员也可以把自己定义的符号常量、宏或函数原型放在头文件中，用 `#include` 命令包含这些头文件。

以下是一个头文件及其应用的例子。两个文件的内容如下：

(1) 文件 `print_format.h`

```
#define PR printf
#define NL "\n"
#define D "%d "
#define D1 D NL
#define D2 D D NL
#define D3 D D D NL
#define D4 D D D D NL
#define S "%s"
```

(2) 文件 `file1.c`

```
#include "print_format.h"
main()
{ int a,b,c,d;
  char string[] = "CHINA";
  a = 1; b = 2; c = 3; d = 4;
  PR(D1,a);
  PR(D2,a,b);
  PR(D3,a,b,c);
  PR(D4,a,b,c,d);
  PR(S,string);
}
```

一个 `#include` 命令只能指定一个被包含文件，如果要包含 n 个文件，就要用 n 个 `#include` 命令。被包含文件中还可以包含别的文件，这被称为嵌套包含。允许的最大嵌套深度随编译程序而变。因此，可以在一个文件中通过多条 `#include` 命令包含多个文件，然后在其他文件中只要用一条 `#include` 命令包含这个文件，即可实现包含多个文件的目的。标准库的很多文件像 `stdio.h` 等都通过这种方式包含了很多文件。

`#include` 命令的文件名，可以使用双引号或尖括号两种形式。括号的类型决定了对指定文件的搜索方式。使用尖括号时，编译程序将按照在编译器中设定的包含目录的顺序搜索，这种方式称为标准方式；使用双引号时，编译程序将先在当前目录搜索，若找不到文件再按标准方式搜索。

被包含文件与其所在的文件，在预处理后成为一个文件，因此，如果被包含文件定义有全局变量，在其他文件中就不必用 `extern` 关键字声明。但一般不在被包含文件中定义变量。有关文件包含命令，在函数与结构一章中，会进行更详细的应用说明。

6.3 条 件 编 译

一般情况下，源程序中的所有行均参加编译，但有时希望部分行在满足一定条件才进行编译，即对部分内容指定编译的条件，称为“条件编译”，它提供并维护有多种定制版本程序的商业软件公司广泛采用条件编译方式。

下面将一一介绍条件编译的指令。

1. #if、#else、#elif、#endif 指令

条件编译指令中使用最多的是#if、#else、#elif和#endif指令。这些指令允许程序员根据常量表达式的结果来决定是否对这些指令包围的代码段进行编译。

1) #if和#endif指令

#if指令的一般形式为：

```
#if 常数表达式  
代码段  
#endif
```

当#if指令后的常量表达式的结果为真时，才对#if和#endif指令之间的代码段进行编译，否则将忽略该代码段。其中，#endif指令用来标识#if块的结束。

现在来看一个例子。程序代码如下：

```
#include <iostream>  
using namespace std;  
#define MAX 200  
void main()  
{  
    #if MAX>150  
        cout<<"本语句被编译了 "<<endl;  
    #endif  
}
```

由于MAX>150，因此cout<<"本语句被编译了"<<endl;语句被执行，故程序输出结果为：

本语句被编译了

2) #else指令

#else指令可以用于在#if后的常量表达式不成立时的备选项。格式如下：

```
#if 常数表达式
    语句段;
#else
    语句段;
#endif
```

上述结构的含义是：若`#if`指令后的常数表达式为真，则编译`#if`到`#else`之间的程序段；否则编译`#else`到`#endif`之间的程序段。

先来看一个例子。程序代码如下：

```
#include <iostream>
using namespace std;
#define MAX 200
void main()
{
    #if MAX>999
        cout<<"语句 1 被编译了"<<endl;
    #else
        cout<<"语句 2 被编译了"<<endl;
    #endif
}
```

程序运行结果如下：

语句 2 被编译了

下面再来看一个例子。

输入一行字母字符，根据需要设置条件编译，使之能将字母全改为大写输出，或全改为小写输出。程序代码如下：

```
#include <iostream>
using namespace std;
#define LETTER 1
void main()
{
    char str[20] = "C++ Language", c;
    int i;
    i = 0;
    while((c=str[i]) != '\0')
    {
        i++;
        #if LETTER
            if (c>='a' && c<='z')
                c = c - 32;
        #else
```



```
        if (c>='A' && c<='Z')
            c = c + 32;
        #endif
        cout<<c;
    }
    cout<<endl;
}
```

程序运行结果如下:

C++ Language

3) #elif 指令

除了#if 和#else 指令外 C++还提供了#elif 指令, 其意思即为"else if", 它与#if 和#else 指令一起构成了 if-else-if 嵌套语句, 用于多种编译选择的情况。其格式一般形式为:

```
#if 常量表达式 1
    程序段 1
#elif 常量表达式 2
    程序段 2
#elif 常量表达式 3
    程序段 3
...
#else
    程序段 n+1
#endif
```

下面举个例子说明#elif 指令的用法。

```
#include <iostream>
using namespace std;
#define A -10
void main( )
{
    #if A>0
        cout<< "a>0"<<endl;
    #elif A<0
        cout<<"a<0"<<endl;
    #else
        cout<< "a==0"<<endl;
    #endif
}
```

程序运行结果如下:

a<0



2. #ifdef、#ifndef、#undef、defined 指令

条件编译的另一个方法是使用编译指令#ifdef和#ifndef, 这两个指令的含义分别是“如果已定义”和“如果未定义”。

1) #ifdef 指令

#ifdef 的一般形式是:

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

它的功能是, 如果标识符已被#define 命令定义过, 则对程序段 1 进行编译; 否则对程序段 2 进行编译。如果没有程序段 2(它为空白), 本格式中的#else 可以没有, 即可以写为:

```
#ifdef 标识符
    程序段
#endif
```

下面来看一个例子。

```
#include <iostream>
using namespace std;
void main()
{
    struct stu
    {
        int num;
        char *name;
        char sex;
        float score;
    } *ps;
    ps=(struct stu*)malloc(sizeof(struct stu));
    ps->num=102;
    ps->name="Zhang ping";
    ps->sex='M';
    ps->score=62.5;
    #ifdef NUM
        cout<<"Number="<<ps->num<<endl<<"Score="<<ps->score<<endl;
    #else
        cout<<"Name="<<ps->name<<endl<<"Sex="<<ps->sex<<endl;
    #endif
    free(ps);
}
```

程序运行结果如下:

```
Name=Zhang ping  
Sex=M
```

由于在程序的第 19 行插入了条件编译预处理命令, 因此要根据 NUM 是否被定义过来决定编译哪一个 cout 语句。而在程序中没有对 NUM 作过宏定义, 因此应对第二个 cout 语句作编译, 故运行结果是输出了姓名和性别。

以下例子用于控制程序调试信息的显示。

```
#define DEBUG  
#ifdef DEBUG  
    cout<<"x="<<x<<" ,y="<<y<<" ,z="<<z<<endl;  
#endif
```

cout 语句被编译, 程序运行时可以显示 x、y、z 的值。若在程序调试完成后, 不再需要显示 x、y、z 的值, 则只需要去掉 DEBUG 标识符的定义。

直接使用 cout 语句显示调试信息, 在程序调试完成后去掉 cout 语句, 也可以达到目的。但如果程序中有很多处需要调试观察, 增、删语句既麻烦又容易出错。而使用条件编译则相当清晰、方便。

2) #ifndef 指令

ifndef 语句的一般形式为:

```
#ifndef 标识符  
    程序段 1  
#else  
    程序段 2  
#endif
```

与第一种形式的区别是将"ifdef"改为"ifndef"。它的功能是, 如果标识符未被#define 命令定义过, 则对程序段 1 进行编译, 否则对程序段 2 进行编译。这与第一种形式的功能正相反。

3) defined 指令

defined 操作符的一般形式如下:

```
defined 宏名
```

如果宏名是当前定义的, 则该表达式为真, 否则为假。

使用#if 和 defined 指令结合, 可以达到和#ifdef 同样的效果。

例如, `#if defined DEBUG` 与 `#ifdef DEBUG` 是相同的含义。

在 `defined` 前加上感叹号 “!” 再配合 `#if` 语句, 可以起到与 `#ifndef` 相同的效果。

以下示例用于: 判断程序的版本。

```
#if ! defined DEBUG
    cout<<"final version" <<endl;
#endif
```

4) `#undef` 指令

`#undef` 指令用来删除事先定义的宏定义, 其一般形式为:

`#undef` 宏名

例如:

```
#define WIDTH 50
...
#undef WIDTH
```

`#undef` 指令主要用于使宏名字局部化, 使其仅局部于需要这类宏名字的代码段。

6.4 其他预处理指令

其他预处理指令包括 `#error`、`#pragma` 以及 `#line`。分别介绍如下。

1. `#error` 指令

`#error` 指令的作用是强制编译程序停止编译, 主要用于程序的调试。`#error` 指令的一般形式为:

```
#error error_message
```

例如: 检测最大值是否定义的程序。其代码如下。

```
#include <iostream>
using namespace std;
void main()
{
    #ifndef MAX
        #error MAX not define
    #endif
}
```

由于 MAX 未定义, 因此#error 语句被编译程序处理, 结束编译, 显示错误信息:

```
fatal error C1189: #error : MAX not define  
g:\VC7\precompile_error\precompile_error.cpp 8
```



注意: 宏串 error_message 不用双引号包围。当遇到#error 指令时, 停止编译并显示错误信息, 同时还可能显示编译程序预先设定的其他内存。

2. #pragma 指令

在所有的预处理指令中, #pragma 指令可能是最复杂的了, 它的作用是设定编译器的状态或者是指示编译器完成一些特定的动作。#pragma 指令对每个编译器都给出了一个方法, 在保持与 C 和 C++ 语言完全兼容的情况下, 给出主机或操作系统专有的特征。依据定义, 编译指示是机器或操作系统专有的, 且对于每个编译器都是不同的。其格式一般为:

```
#pragma Para
```

其中 Para 为参数。但是, 不同的编译器实现的参数有很大的不同。在实际工作中, 要尽量减少利用该语句加参数的形式来改变编译器的工作状态, 因为这样做可能会导致不同平台下编译出的程序有很大的差别。

下面来看一些常用的参数。

1) message 参数

message 参数是最方便的一个参数, 它能够在编译信息输出窗口中输出相应的信息, 这对于源代码信息的控制是非常重要的。其使用方法为:

```
#pragma message("消息文本")
```

当编译器遇到这条指令时就在编译输出窗口中将消息文本打印出来。

当程序中定义了许多宏来控制源代码版本的时候, 设计者自己都有可能会忘记有没有正确地设置这些宏, 此时可以用这条指令在编译的时候就进行检查。假设希望判断有没有在源代码的什么地方定义了_X86 这个宏, 可以用下面的方法:

```
#ifndef _X86  
#pragma message("_X86 macro activated!")  
#endif
```

当定义了_X86 这个宏以后, 应用程序在编译时就会在编译输出窗口里显示“_X86 macro activated!”。这样就不会因为不记得是否定义过一些特定的宏而困惑了。

2) code_seg 参数

格式如下:

```
#pragma code_seg( ["section-name" [, "section-class"] ] )
```

它能够设置程序中函数代码存放的代码段，开发驱动程序的时候就会经常使用到它。

3) #pragma once

只要在头文件的最开始加入这条指令就能够保证头文件被编译一次。这条指令实际上在 VC6 中就已经有了，但是考虑到兼容性，故并没有太多地使用它。

4) #pragma hdrstop

该指令表示预编译头文件到此为止，后面的头文件不进行预编译。Borland C++ Builder(BCB)可以预编译头文件，以加快链接的速度，但如果所有的头文件都进行预编译，又可能占太多磁盘空间，所以使用这个选项来排除一些头文件。

有时单元之间有依赖关系，比如单元 A 依赖单元 B，所以单元 B 要先于单元 A 编译。你可以用 #pragma startup 指定编译优先级，如果使用了 #pragma startup 指令，BCB 就会根据优先级的大小先后编译。

5) #pragma resource "*.dfm"

该指令表示把 *.dfm 文件中的资源加入工程。*.dfm 中包括窗体外观的定义。

6) #pragma warning

该指令用来控制警告信息的显示。其参数可以是一个，也可以是多个。例如：#pragma warning(disable:4507 34; once:4385; error:164) 等价于：

```
#pragma warning(disable:4507 34)    // 不显示 4507 和 34 号警告信息
#pragma warning(once:4385)          // 4385 号警告信息仅报告一次
#pragma warning(error:164)          // 把 164 号警告信息作为一个错误
```

同时这个 #pragma warning 指令也支持如下格式：

```
#pragma warning( push [ , n ] )
#pragma warning( pop )
```

这里 n 代表一个警告等级(1~4)。

```
#pragma warning( push ) //保存所有警告信息现有的警告状态
#pragma warning( push, n) //保存所有警告信息现有的警告状态，并且把全局警告等级设定为 n
```

#pragma warning(pop)向栈中弹出最后一个警告信息，在入栈和出栈之间所作的一切改动取消。例如：

```
#pragma warning( push )
#pragma warning( disable : 4705 )
#pragma warning( disable : 4706 )
```

```
#pragma warning( disable : 4707 )  
//...  
#pragma warning( pop )
```

在这段代码的最后，重新保存所有的警告信息(包括 4705、4706 和 4707)。

7) pragma comment(...)

该指令将一个注释记录放入一个对象文件或可执行文件中。常用的 lib 关键字，可以帮助应用程序连入一个库文件。

3. #line 指令

#line 指令告诉预处理器将编译器内部存储的行号和文件名转变为一个给定的行号和文件名。编译器使用该行号和文件名指出编译过程中发现的错误。行号一般指的是当前输入行，文件名指当前输入文件。每处理一行，行号就增 1。

#line 指令的语法如下。

```
#line 数字序列 "文件名"opt
```

数字序列的值可以是任何整型常数。文件名可以是任意字符的组合，且应用双引号括起来。如果省略文件名，则前面的文件名保持不变。

可以通过编写一个#line 指令来改动源行号和文件名。翻译器使用行号和文件名来确定预定义宏 __FILE__ 和 __LINE__ 的值。你可以使用这些宏把自描述错误消息加入到程序文本中。有关这些宏的更多信息参见预定义的宏。

__FILE__ 宏扩展成内容为用双引号括起的文件名的一个字符串。

如果改变行号和文件名，编译器将忽略原有的值，用新值继续处理。#line 指令通常被程序生成器用来生成指向最初源程序的错误消息，而不是生成程序。下面的例子用于说明#line 以及 __LINE__ 和 __FILE__ 宏。在这个语句中，内部存储的行号设置为 151，文件名改为 copy.c。

```
#line 151 "copy.c"
```

在下面的例子中，若一个给定的“断言”(assertion)不为真，则宏 ASSERT 使用预定义宏 __LINE__ 和 __FILE__ 打印出一个关于源文件的错误消息。

```
#define ASSERT(cond)  
if( !(cond) ) \  
{  
    cout<<"assertion error line "<<__LINE__<<"",  
file("<<__FILE__<<")"<<endl;  
}
```

本章小结

本章主要介绍宏和编译预处理，主要内容包括：宏的定义及使用；编译预处理；条件编译；以及更多的编译预处理指令。

习 题

1. 下列程序执行后，其输出的值是多少？

```
#define a 30
#define b a+3
#define c b*3
void main()
{
    int x = c*3;
    cout<<x<<endl;
}
```

2. 设计一个网络通信程序，要求可以在 Windows 和 Linux 都能编译运行通过。
(本题目超出 C++ 的范围，要求学生自学对应的系统 API)



第 7 章 数 组

本章内容：

- 数组的定义和使用。
- 数组下标的使用。
- 排序算法。

重点：

数组的定义和使用，排序算法。

目的：

利用数组来记录数据和处理数据，了解算法的初步知识，提高程序设计的技巧。

C++提供了很多的基本数据类型。前面已经介绍过的数据类型有整型(int、unsigned int、short int、long int、unsigned short、unsigned long 等)、实型(float、double 和 long double 等)、字符型(char、unsigned char 等)和指针等。这些基本数据类型也被称为原子类型，意思是它们具有原子特性，不可以再分割。它们是由 C++语言规范预先定义好的。

本章将介绍数组的概念。数组是由基本数据类型、用户自定义数据类型(比如后面要讲的“类”)组合而成的一种数据的组织结构，亦即一种数据结构。形象来讲，数组就像是一片连排的教室，由多个单独的教室组成。本章将着重介绍一维和多维数组的声明和初始化，以及数组在数据结构中的重要应用。

7.1 为何需要数组

在写程序时，常常需要对大量同种类型的数据进行处理，如排序、查找等操作。对于这种情况，为每个需要处理的数据都声明一个变量是一件非常令人头疼的事情，比如若要对 500 个 int 型的数据进行排序，为这 500 个数声明 500 个变量几乎是不可能的。起 500 个变量名问题并不是很大，但要想仔细记清每个变量名对应哪个数据就比较麻烦了，更不用说每一次处理都要手工编写代码的痛苦了，程序员的所有精力都要投入在应付这几个变量名上，而无法去关注真正值得关心的算法问题。

C++中可以通过声明数组来解决此问题,数组是具有一定顺序关系的若干相同类型变量的集合体,组成数组的变量称为该数组的元素。通过使用数组的下标可以很容易地操作数组中的数据。

7.2 声明数组

在 C++中,允许使用同一个名字来命名一组在内存当中依次存放的同种数据类型的数据,这个名字即为一个数组。例如要表示 100 个人的编号,方法一是单独声明 100 个变量来表示这 100 个编号;方法二是使用数组来表示,在这里使用 `people` 来表示数组。而对每一个数据本身,用数组名加上写在方括弧中的序号(称为下标)标示,即 `people [0]`, `people [1]`, `people [2]`, ..., `people [99]`, 这些即为数组元素。这些数组元素在内存空间中是按顺序放在一个连续的空间的,所以访问效率很高。

一维数组的一般说明形式如下:

```
type_specifier var_name [size];
```

`type_specifier` 用来指定数组中数据的类型,该类型既可以是基本数据类型,也可以是用户定义的结构或类。

`var_name` 指定数组的名字,同普通变量声明一样,数组名的选取遵循 C++关于标识符的一般规定。

`size` 指定数组的大小,表示数组元素的个数。在 C++中, `size` 在声明时必须为一个整数常量(或值为整数常量的表达式),编译器需要根据此常量来确定需要为数组申请多大的存储空间。这一点不像 VB 等程序设计语言,在 VB 中,数组的大小是可以动态变化的,而且可以在声明以后重新指定,但是在 C++中不允许这样。若需要动态数组,需要通过 `malloc` 或 `new` 指令动态地申请内存来实现,这些将在以后章节中讲到。

例如:

```
①int idata[100];  
②class T;  
T tdata[20];
```

在上面的第一个例子中,声明了一个大小为 100 的整型数组,编译器会在编译期间申请到 100 个 `int` 型大小的存储空间。

同理,第 2 个例子声明的是类型为类 T 的数组,大小为 20 个 T 型数据。

下面的例子是错误的。

```
int i;  
double ddata[i];
```

这个例子声明数组的方式是错误的，数组大小必须由常量给出，在本例中试图通过变量 *i* 来动态申请数组，在编译时系统会提示错误。

声明数组的另一种方式是：数组元素的个数并不直接给出，而是通过在声明时给出的元素初值的个数来确定。

例如：

```
char ch[]={'h','e','l','l','o'};
```

该字符数组 *ch* 由 5 个字符元素组成，元素个数在编译时，编译器会自动计算得到。上面的声明等价于：

```
char ch[5]={'h','e','l','l','o'};
```

但是，若需要的数组元素个数大于初始化给出的元素个数时，元素的个数必须提前给出。

例如：

```
char ch[10]={'h','e','l','l','o'};
```

本例声明了一个由 10 个字符元素组成的数组，前 5 个元素的值为 {'h','e','l','l','o'}。

7.3 访问数组元素

7.2 节说明了如何去声明一个数组，那么如何去访问数组中任意的元素呢，答案是：数组下标。通过数组的下标可以访问各个数组元素，对于个数为 *N* 的数组，各个元素的下标从 0 开始，依次加 1，最后一个元素的下标为 *N*-1。例如：

```
#include <iostream>  
using namespace std;  
void main()  
{  
    int idata[10];  
    int i;  
    for(i = 0; i < 10; i++)  
    {  
        idata[i] = i ; //为第 i+1 个元素赋值，其下标为 i  
    }  
    cout<<"数组中的 10 个元素为: "<<endl;
```



```

    for(i = 0;i < 10;i++)
    {
        cout<<idata[i]<<' '; //输出第 i+1 个元素赋值，其下标为 i
    }
}

```

在上面的例子中，通过数组下标的变化对每个数组元素依次赋值，然后输出每个元素的值。

程序运行结果如下：

数组中的 10 个元素为：

1 2 3 4 5 6 7 8 9 10

注意，C++并不检查数组下标的有效性，数组的前后两端都有可能出现越界的现象。对数组下标有效性的维护由程序员来负责。若对数组元素进行操作，而数组下标跃出了数组的边界时，并不会给出任何错误提示，但此时访问的内存空间已经错误了，有时甚至会修改其他程序的数据，这对系统的稳定性来说是非常可怕的。例如：

```

#include <iostream>
using namespace std;
void main()
{
    int idata[10];
    int i;
    for(i = 1;i <= 10;i++)
    {
        idata[i] = i ; //为第 i+1 个元素赋值，其下标为 i，最后一个会越界
    }
    cout<<"数组中的 10 个元素为： "<<endl;
    for(i = 0;i < 10;i++)
    {
        cout<<idata[i]<<' '; //输出第 i+1 个元素赋值，其下标为 i
    }
}

```

在上面的例子中，由于 idata 的个数为 10，数组的最大下标应为 9，idata[10]使用了越界的下标，赋值的地址虽然紧挨上一个元素，但是由于它不是程序开辟的正常内存，故其对其他数据的影响将无法预料。可能程序输出结果仍然相同，没有人会注意到这次越界；也有可能错误地修改了其他程序的数据；更严重的也许立即导致系统崩溃。

7.4 数组的初始化

数组初始化的方式有声明时初始化和使用时初始化两种。^①

1. 声明时初始化

对数组的声明是定义性的声明，在声明的同时可以对数组各元素初始化，初始化表达式按元素顺序依次写在一对花括号内。也即在数组声明时就给出数组元素的初值。例如：

```
int idata[5]={1,2,3,4,5}
```

在初始化时，也可以只给一部分元素赋初值。对于不需初始化的值可以通过“,”符号略过，例如：

```
int a[10]={0,1,2,3,4};
```

//定义了一个10个元素的数组并给出了前5个元素的初值

```
int b[10]={,1,,3,,5};
```

//定义了一个10个元素的数组并对第2,4,6个元素进行了初始化，不需初始化的元素通过','符号略过

声明时初始化也可以不指定数组的大小，编译系统会根据初始值的个数自动决定数组的大小。

2. 使用时初始化

使用时初始化是指声明时只指定数组元素的个数，使用前才进行元素的初始化。例如：

```
#include <iostream>
```

```
using namespace std;
```

```
void main()
```

```
{
```

```
    int idata[10];
```

```
    int i;
```

```
    for(i = 0;i < 10;i++)
```

```
    {
```

```
        idata[i] = i ; //为第i+1个元素赋值，其下标为i
```

```
    }
```

```
    cout<<"数组中的10个元素为："<<endl;
```

```
    for(i = 0;i < 10;i++)
```

① 实际上，还有第三种初始化方式：若声明的数组为全局数组或静态数组，则即使在声明时不显式地给出元素的初值，数组的各个元素也会被系统自动地初始化为0。

```

    {
        cout<<idata[i]<<' '; //输出第 i+1 个元素赋值, 其下标为 i
    }
}

```

程序先声明了 10 个整型元素组成的数组, 然后在后面的程序中进行了初始化。

7.5 数组应用举例

数组中存放的是同种类型的数据, 对这些数据进行排序是对数组进行的最常见的操作, 在接下来的几节中将详细介绍几种比较经典的排序算法。为了说明方便, 定义如下数组: `int array[10]`, 要求做的工作是从大到小进行排序。

7.5.1 选择排序

1. 基本的选择排序

1) 基本思想

首先从要排序的数中选择最大的数, 将它放在第一个位置, 然后从剩下的数中选择最大的数放在第二个位置, 如此继续, 直到最后从剩下的两个数中选择最大的数放在倒数第二个位置, 剩下的一个数放在最后位置, 完成排序。下面是六个元素排序的过程。

4 5 7 1 2 3	//4 和 5 比较, 4<5, 交换位置
5 4 7 1 2 3	//5 和 7 比较, 5<7, 交换位置
7 4 5 1 2 3	//7 和 1 比较, 7>1, 位置不变
7 4 5 1 2 3	//7 和 2 比较, 7>2, 位置不变
7 4 5 1 2 3	//7 和 3 比较, 7>3, 位置不变。第一趟排序结束
⑦ 4 5 1 2 3	//4 和 5 比较, 4<5, 交换位置
7 5 4 1 2 3	//5 和 1 比较, 5>1, 位置不变
7 5 4 1 2 3	//5 和 2 比较, 5>2, 位置不变
7 5 4 1 2 3	//5 和 3 比较, 5>3, 位置不变, 第二趟排序结束
7 ⑤ 4 1 2 3	//4 和 1 比较, 4>1, 位置不变

7	5	4	1	2	3	//4 和 2 比较, 4>2, 位置不变
		<u> </u>				
7	5	4	1	2	3	//4 和 3 比较, 4>3, 位置不变, 第三趟排序结束
		<u> </u>				
7	5	④	1	2	3	//1 和 2 比较, 1<2, 交换位置
		<u> </u>				
7	5	4	2	1	3	//2 和 3 比较, 2<3, 交换位置, 第四趟排序结束
		<u> </u>				
7	5	4	③	1	2	//1 和 2 比较, 1<2, 交换位置, 第五趟排序结束
		<u> </u>				
7	5	4	3	②	①	

2) 算法实现

```
for(int i=1;i<9;i++)
    for(int j=i+1;j<10;j++)
    {
        if(a[i]<a[j])
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
    }
```

2. 改进

以上排序方案每次交换两个元素都需要执行三个语句, 过多的交换必定要花费许多时间。改进方案是在内循环的比较中找出最大值元素的下标, 在内循环结束时才考虑是否要调换。

代码如下:

```
for(int i=0;i<9;i++)
{
    k = i;
    for(j = i+1;j<20;j++)
        if(a[j]>a[k])
            k = j;
        if(i < k)
        {
            temp = a[i];
            a[i] = a[k];
            a[k] = temp;
        }
}
```



7.5.2 冒泡排序

1. 基本的冒泡排序

1) 基本思想

依次比较相邻的两个数，把大的放前面，小的放后面。即首先比较第1个数和第2个数，大数放前，小数放后。然后比较第2个数和第3个数……直到比较完最后两个数。第一趟结束，最小的一定沉到最后。重复以上过程，仍从第1个数开始，到倒数第2个数。然后……由于在排序过程中总是大数往前，小数往后，相当于气泡上升，所以叫冒泡排序。下面是6个元素排序的过程。

4 5 7 1 2 3	//4 和 5 比较, $4 < 5$, 交换位置
<u>4</u> 5 7 1 2 3	
5 4 7 1 2 3	//4 和 7 比较, $4 < 7$, 交换位置
<u>5</u> 4 7 1 2 3	
5 7 4 1 2 3	//4 和 1 比较, $4 > 1$, 位置不变
<u>5</u> 7 4 1 2 3	
5 7 4 1 2 3	//1 和 2 比较, $1 < 2$, 交换位置
<u>5</u> 7 4 1 2 3	
5 7 4 2 1 3	//1 和 3 比较, $1 < 3$, 交换位置, 第一趟排序结束
<u>5</u> 7 4 2 3 ①	
5 7 4 2 3 ①	//5 和 7 比较, $5 < 7$, 交换位置
<u>5</u> 7 4 2 3 ①	
7 5 4 2 3 1	//5 和 4 比较, $5 > 4$, 位置不变
<u>7</u> 5 4 2 3 1	
7 5 4 2 3 1	//4 和 2 比较, $4 > 2$, 位置不变
<u>7</u> 5 4 2 3 1	
7 5 4 2 3 1	//2 和 3 比较, $2 < 3$, 交换位置, 第二趟排序结束
<u>7</u> 5 4 2 3 1	
7 5 4 3 ② 1	//7 和 5 比较, $7 > 5$, 位置不变
<u>7</u> 5 4 3 ② 1	
7 5 4 3 2 1	//5 和 4 比较, $5 > 4$, 位置不变
<u>7</u> 5 4 3 2 1	
7 5 4 3 2 1	//4 和 3 比较, $4 < 3$, 位置不变, 第三趟排序结束
<u>7</u> 5 4 3 2 1	
7 5 4 ③ 2 1	//7 和 5 比较, $7 > 5$, 位置不变
<u>7</u> 5 4 ③ 2 1	
7 5 4 3 2 1	//5 和 4 比较, $5 < 4$, 位置不变, 第四趟排序结束
<u>7</u> 5 4 3 2 1	
7 5 ④ 3 2 1	//7 和 5 比较, $7 > 5$, 位置不变, 第五趟排序结束
<u>7</u> 5 ④ 3 2 1	
⑦ ⑤ 4 3 2 1	

2) 算法实现

```
for(int i=0;i<9;i++)
    for(j=0;j<10-i;j++)
    {
        if(a[j]<a[j+1])    //如果某一数据小于后面的数据
        {
            temp = a[j];
            a[j] = a[j+1];    //执行交换
            a[j+1] = temp;
        }
    }
}
```

2. 改进

上例中的第二趟结束时已经排好序。但是计算机此时并不知道已经排好了。所以还需进行一次比较。如果没有发生任何数据交换，则知道已经排好序，可以不再比较了。因此第三趟比较还需进行，第四趟、第五趟比较则不必要。这里设置一个布尔变量 **bo**，记录是否进行了比较。值为 **false** 表示进行了比较，**true** 则表示没有。代码如下：

```
i = 1;
do
{
    bo = true;
    for(j = 0;j<10-i;j++)
    {
        if(a[j]<a[j+1])
        {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
            bo = false;
        }
    }
}
i++;
while(bo);    //如果已经不存在交换事件，则意味着不需要再比较了
```

3. 再次改进

如果有 20 个元素，数据序列是 8, 3, 4, 9, 7 再后跟着 15 个大于 9 且已经排好序的数据，在第三趟后算法终止，总共做了 $19+18+17=54$ 次比较，使得绝大多数已排好序的数据在一遍扫描后又被检查 3 遍。程序改进如下：

```
flag = 9;
```

```

while(flag>0)
{
    k = flag-1;
    flag = 0;
    for(i=0;i<k;i++)
    {
        if(a[i]<a[i+1])
        {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
            flag = i;
        }
    }
}

```

改进的冒泡算法对上述数据进行的比较次数是 $19+4+2=24$ 。

7.5.3 更多排序算法

1. 希尔排序法

基本思想：希尔排序法，又称减少增量的排序，是 1959 年由 D.L.Shell 提出来的。图 7-1 所示为以 8 个元素排序示范的例子。在该例中，开始时相隔 4 个成分，分别按组进行排序，这时每组 2 个成分，共 4 组；然后相隔 2 个成分，再按组排序……最后对所有相邻成分进行排序。

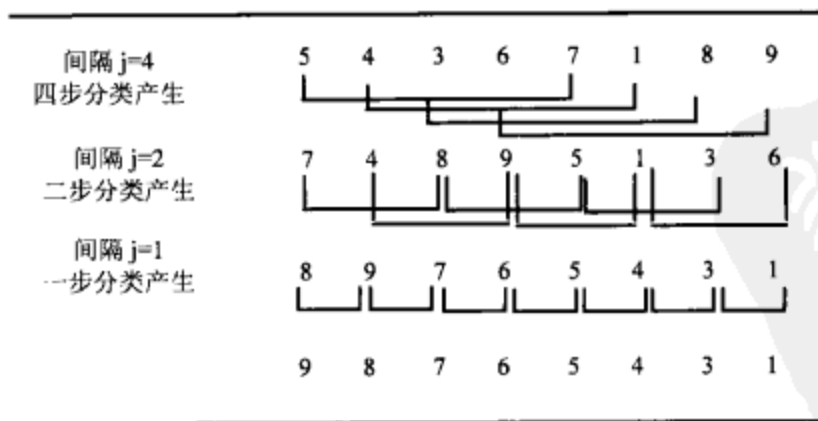


图 7-1 希尔排序过程演示

2. 插入排序

基本思想：插入排序的思想就是读一个，排一个。将第 1 个数放入数组的第 1 个元素中，以后读入的数与已存入数组的数进行比较，确定它在从大到小的排列中应处的位置。

将该位置以及以后的元素向后推移一个位置，再将读入的新数填入空出的位置中。

3. 合并排序

基本思想：合并排序的算法就是二分法。

合并排序的步骤如下。

- (1) 分解：将 n 个元素分解成各含一半元素的子序列。
- (2) 解决：用合并排序法对两个子序列递归地排序。
- (3) 合并：合并两个已排序的子序列排序结果。

在对子序列排列时，当其长度为 1 时递归结束，因为单个元素被认为是已排好序的。合并排序的关键步骤在于合并目前产生的两个已排好序的子序列。

若要将 $A[p..q]$ 和 $A[q+1..r]$ 合并成一个已排好序的子序列 $A[p..r]$ ，可引入一个辅助过程 $\text{merge}(A, p, q, r)$ 来完成这一项合并工作。其中 A 是数组， p 、 q 、 r 是下标。

4. 快速排序^①

- (1) 算法思想。

快速排序是 C.R.A.Hoare 于 1962 年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。

- (2) 分治法的基本思想。

分治法的基本思想是：将原问题分解为若干个规模更小但结构与原问题相似的子问题，然后递归地解决这些子问题，再将这些子问题的解组合为原问题的解。

- (3) 快速排序的基本思想。

设当前待排序的无序区为 $R[\text{low}]..R[\text{high}]$ ，利用分治法可将快速排序的基本思想描述如下。

① 分解：在 $R[\text{low}]..R[\text{high}]$ 中任选一个记录作为基准(pivot)，以此基准将当前无序区划分为左、右两个较小的子区间 $R[\text{low}]..R[\text{pivotpos}-1]$ 和 $R[\text{pivotpos}+1]..R[\text{high}]$ ，并使左边子区间中所有记录的关键字均小于等于基准记录(不妨记为 pivot)的关键字 pivot.key ，右边的子区间中所有记录的关键字均大于等于 pivot.key ，而基准记录 pivot 则位于正确的位置(pivotpos)上，它无须参加后续的排序。

① 快速排序算法，需要利用函数递归来完成，因此本节内容可以在学习完函数一章后再深入地学习，或在数据结构课程中进行深入学习。



注意：划分的关键是要求出基准记录所在的位置 `pivotpos`。划分的结果可以简单地表示为(注意 `pivot=R[pivotpos]`):

$R[low]..R[pivotpos-1].keys \leq R[pivotpos].key \leq R[pivotpos+1]..R[high].keys$

其中 $low \leq pivotpos \leq high$ 。

② 求解：通过递归调用快速排序，对左、右子区间 $R[low]..R[pivotpos-1]$ 和 $R[pivotpos+1]..R[high]$ 进行快速排序。

③ 组合：因为当“求解”步骤中的两个递归调用结束时，其左、右两个子区间已有序。对快速排序而言，“组合”步骤无须做什么，可看作是空操作。

2) 算法实现

```
void QuickSort(SeqList R, int low, int high)
{
    //对 R[low]..R[high] 快速排序
    int pivotpos;           //划分后的基准记录的位置
    if(low<high)            //仅当区间长度大于1时才须排序
    {
        pivotpos=Partition(R, low, high); //对 R[low]..R[high] 做划分
        QuickSort(R, low, pivotpos-1);    //对左区间递归排序
        QuickSort(R, pivotpos+1, high);    //对右区间递归排序
    }
}                           //QuickSort
```

7.6 字符串与字符数组

字符串是字符数组中的一种特例，数组的最后一个元素为空字符 `'\0'`。因此，作为字符串使用的数组比一般的字符数组要至少多一个结束符，如图 7-2 和图 7-3 所示。

h	e	l	l	o
---	---	---	---	---

图 7-2 普通字符数组

h	e	l	l	o	\0
---	---	---	---	---	----

图 7-3 字符串数组

先看一个例子。程序代码如下：

```
#include <iostream.h>
void main()
{
    char ch[5] = {'h','e','l','l','o'}, str[6] = {'h','e','l','l','o','\0'};
```

```

    for(int i=0; i < 5; i++)
    {
        cout<<ch[i];
    }
    cout<<endl;
    cout<<str<<endl;
}

```

程序运行结果如下:

```

hello
hello

```

在本例中, 声明了一个普通的字符数组和一个作为字符串使用的数组, 可以看到作为字符串使用的数组比普通数组多一个结束符。

在用 `cout` 语句输出时, 作为字符串使用的数组可以直接由 `cout<<str` 输出, `cout` 通过 `'\0'` 判断输出结束的位置。

但普通字符数组由于没有结束符, 无法判断输出结束位置, 必须对每个元素循环输出。

将上例改写如下:

```

#include <iostream.h>
void main()
{
    char ch[5] = {'h','e','l','l','o'}, str[6] = {'h','e','l','l','o','\0'};

    cout<<ch<<endl;
    cout<<str<<endl;
}

```

输出 `ch` 时并不是输出 `hello` 就结束, 而是继续对与该数组相邻内存的内容进行输出, 直到发现 `'\0'` 为止, 显然此时对数组的操作已经越界。

常用的字符串处理函数及其功能如表 7-1 所示。

表 7-1 常用的字符串处理函数及其功能

函 数	功 能
<code>strcpy(s1, s2)</code>	将 <code>s2</code> 复制到 <code>s1</code>
<code>strcat(s1, s2)</code>	将 <code>s2</code> 连接到 <code>s1</code> 的末尾
<code>strlen(s1)</code>	返回 <code>s1</code> 的长度
<code>strcmp(s1, s2)</code>	若 <code>s1</code> 与 <code>s2</code> 相等, 返回值为 0 若 <code>s1 < s2</code> , 返回值小于 0 若 <code>s1 > s2</code> , 返回值大于 0

下面是几个字符串处理函数的应用示例。

```
#include <iostream.h>
main()
{
    char s1[80], s2[80];    //定义字符数组
    gets(s1);              //输入字符串
    gets(s2);
    cout<<"lengths: "<<strlen(s1)<<strlen(s2)<<endl;
    if (!strcmp(s1, s2))
        cout<<"the strings are equal"<<endl;
    strcat(s1, s2);
    cout<<s1;
}
```

运行程序并以“hello”和“hello”这两个串作为输入时，其输出为：

```
hello
hello
lengths:5 5
The strings are equal
hellohello
```

7.7 数组作为函数参数^①

在前面的章节中，使用函数中的参数均为基本数据类型的单个变量。对于数组，也可以作为函数的参数来使用。

下面的程序把有 10 个元素的数组用冒泡排序法进行升序排列。此程序使用数组作为函数参数。代码如下：

```
#include <iostream.h>
void bubble(int[],int);
void main()
{
    int array[]={55,2,6,4,32,12,9,73,26,37};
    int len=sizeof(array)/sizeof(int);

    for(int i =0; i <len; i ++)
```

① 函数以及函数的参数，在第 8 章中才进行深入的讲解，因此建议读者在学习完第 8 章后，再回过头来学习此节。之所以不把函数的内容讲完了再讲这一章，是因为这两章节的内容是交错的，不管将哪个章节放在前面，都会产生同样的问题。而将数组的内容放在前面，对读者来说，需要交换阅读次序的是少部分。反之在函数中，却大量需要使用数组。

```

        cout<<array[i]<< ", ";

        cout<<endl<<endl;
        bubble(array,len);
    }
    void bubble(int a[],int size)
    {
        int i,temp;
        for (int pass=1;pass<size;pass++)
        {
            for(i =0; i <size-pass; i ++ )
                if(a[i]>a[i +1])
                {
                    temp=a[i];
                    a[i]=a[i +1];
                    a[i +1]=temp;
                }
        }
        for(i =0; i <size; i ++ )
            cout<<a[i]<< ", ";
        cout<<endl;
    }
}

```

7.8 二维数组

7.8.1 二维数组的定义

二维数组定义的一般形式为：

类型说明符 数组名[常量表达式][常量表达式]

例如：

float a[3][4]; //a 为 3×4 (3 行 4 列) 的数组
float b[5][10]; //b 为 5×10 (5 行 10 列) 的数组

二维数组的理解，如图 7-4 所示。

二维数组 a[3][4] 理解为： 有三个元素 a[0]、a[1]、a[2]，每一个 元素是一个包含 4 个元素的数组	$a \begin{bmatrix} a[0] & \text{-----} & a_{00} & a_{01} & a_{02} & a_{03} \\ a[1] & \text{-----} & a_{10} & a_{11} & a_{12} & a_{13} \\ a[2] & \text{-----} & a_{20} & a_{21} & a_{22} & a_{23} \end{bmatrix}$
---	---

图 7-4 二维数组的理解示意

二维数组的元素在内存中的存放顺序如图 7-5 所示。

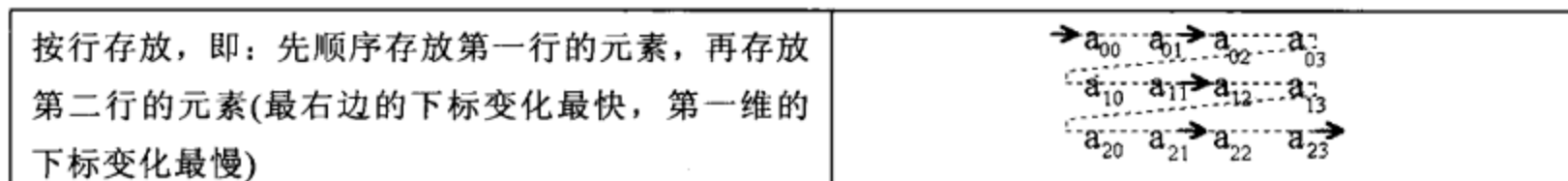


图 7-5 二维数组的元素在内存的存放顺序

7.8.2 二维数组中元素的引用

用数组名和下标引用元素。例如：`float a[2][3]`；有 6 个元素，可按如下方式引用各元素：`a[0][0]`、`a[0][1]`、`a[0][2]`、`a[1][0]`、`a[1][1]`、`a[1][2]`。



注意：数组 `float a[2][3]` 中无元素 `a[2][3]` (下标从 0 开始)。

7.8.3 二维数组的初始化

数组有四种初始化方法，分别介绍如下。

1. 分行赋值

示例如下：

```
static int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

2. 全部数据写在一个大括号内

示例如下：

```
static int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

3. 部分元素赋值

示例如下：

```
static int a[3][4] = {{1},{5},{9}};
```

此句仅对 `a[0][0]`、`a[1][0]` 和 `a[2][0]` 赋值，其余元素未赋值(对于静态数组，编译器自动为未赋值元素指定初值 0；对于动态数组，未赋值元素的初值是随机的)。

4. 其他情况

如果对全部元素赋初值，则第一维的长度可以不指定，但必须指定第二维的长度。例如：

```
static int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

与下面定义等价:

```
static int a[ ][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

7.8.4 二维数组程序举例

例 1 将下面的一个二维数组 a 进行行和列交换, 然后存到另一个二维数组 b 中。
例如:

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad b = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

算法: $b[j][i] = a[i][j]$

程序:

```
main()
{
    static int a[2][3] = {{1,2,3},{4,5,6}};
    static int b[3][2], i, j;
    printf("array a:\n");
    for(i=0; i<=1; i++)    //0~1 行
    {
        for(j=0; j<=2; j++)    //0~2 列
        {
            printf("%5d", a[i][j]);
            b[j][i] = a[i][j];    //行、列交换
        }
        printf("\n");    //输出一行后换行
    }
    printf("array b:\n");
    for(i=0; i<=2; i++)
    {
        for(j=0; j<=1; j++)
            printf("%5d", b[i][j]);
        printf("\n");    //输出一行后换行
    }
}
```

例 2 有一个 3×4 的矩阵, 要求编程序求出其中值最大的那个元素的值及其所在的行号和列号。

分析: 首先把第一个元素 $a[0][0]$ 作为临时最大值 max, 然后把临时最大值 max 与每一个元素 $a[i][j]$ 进行比较, 若 $a[i][j] > \max$, 则把 $a[i][j]$ 作为新的临时最大值, 并记录下其下标 i 和 j。当全部元素比较完后, max 则是整个矩阵全部元素的最大值。

其流程如图 7-6 所示。

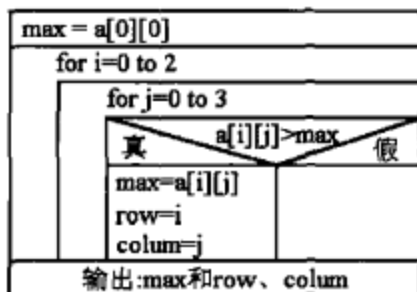


图 7-6 程序流程

程序:

```

main()
{
    int i,j,row=0,column=0,max;
    static int a[3][4]={{1,2,3,4},{9,8,7,6},{-10,10,-5,2}};
    max = a[0][0];
    for(i=0; i<=2; i++) //用两重循环遍历全部元素
        for(j=0; j<=3; j++)
            if (a[i][j] > max )
                { max = a[i][j];
                  row = i;
                  column = j;
                }
    printf("max=%d, row=%d, column=%d\n",max,row,column);
}
  
```



注意: 本例中得到的行列值从 0 开始。

说明: 输入 4 名学生数学、物理、英语、化学、C++五门课的考试成绩, 求出每名学生的平均分, 打印出表格。

分析: 用二维数组 `a` 存放所给的数据, 第一个下标表示学生的学号, 第二个下标表示该学生的某科成绩, 如 `a[i][1]`、`a[i][2]`、`a[i][3]`、`a[i][4]`、`a[i][5]` 分别存放第 `i` 号学生数学、物理、英语、化学、C++五门课的考试成绩, 由于要求每个学生的总分和平均分, 所以第二下标可多开两列, 分别存放每个学生 5 门成绩的总分及平均分。

源程序如下:

```

#include<iostream>
using namespace std;
float result[4,7];

void main()
  
```

```

{
    int i,j;
    cout<<"Enter 4 students score"<<endl;
    for(i=0;i<4;i++)
    {
        a[i][6] = 0;
        for(j=0;j<5;j++)
        {
            cout<<result[i][j];
            result[i][5] += result[i][j];
        }
        result[i][6] = result[i][5] / 5.0;
    }
    cout<<" No.Mat. Phy.Eng. Che. Pas.Tot.Ave."<<endl;
    for(i=0;i<4;i++)
    {
        for(j=0;j<7;j++)
        {
            cout<<result[i][j];
            result[i][5] += result[i][j];
        }
        cout<<endl;
    }
}

```

7.9 多维数组

将数组的概念进一步推广，当数组的维数超过三维时，即构成多维数组。多维数组的一般说明形式为：

类型说明符 数组名 [常量表达式 1] [常量表达式 2] [常量表达式 2]... [常量表达式 n]

多维数组与一维数组和二维数组的唯一区别是其维数的多少。如果把一维数组想象成一张线性表，则二维数组是一个平面矩阵，三维数组则构成了一个空间立方体(如图 7-7 所示)，以此类推。

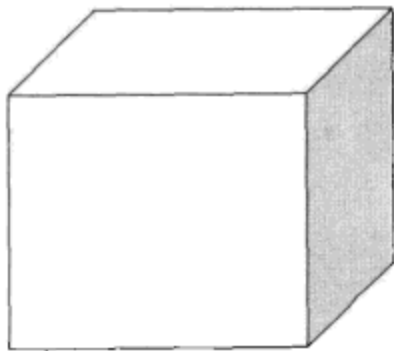


图 7-7 多维数组示意

每一维都可以认为是该立方体中的一个坐标。

事实上，多维数组在内存中的存储仍然是线性存储。

例如，三维数组 $a[2][2][2]$ 的存储格式如图 7-8 所示。

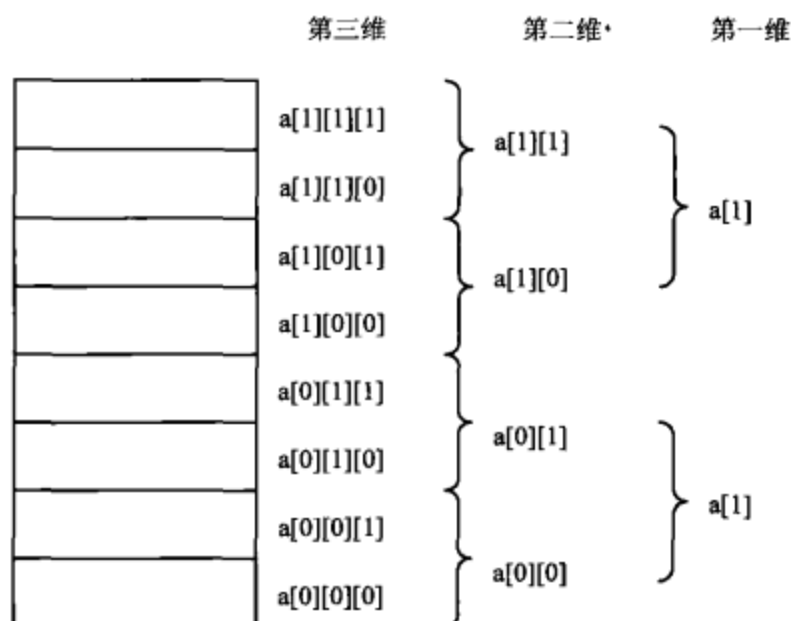


图 7-8 $a[2][2][2]$ 的存储格式

对其在空间中展开便成为前面看到的立方体。对于文档管理比较熟悉的人也可以把它想象成一个多级索引。

7.9.1 多维数组的定义

多维数组的定义格式如下：

数据类型 多维数组名 [常量表达式 1] [常量表达式 2] [常量表达式 3] ... [常量表达式 n]

例如：

```
int a[64][64][3];
const C = 36;
double b[C][C][C][10];
```

7.9.2 多维数组的引用

多维数组的引用与一维数组引用类似，依然是采用下标的形式。引用方法为：

多维数组名 [下标 1] [下标 2] [下标 3] ... [下标 n]

多维数组的下标同一维数组和二维数组的一样，每一维都是从 0 开始，到 $N-1$ 结束，其中 N 是每一维元素的个数。

以下示例引用了一个多维数组程序代码如下:

```
#include <iostream>
using namespace std;
void main()
{
    int a[3][3][3];

    for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
            for(int k=0;k<3;k++)
                {
                    a[i][j][k]=rand();    //初始化为随机数
                }
    for(int i=0;i<3;i++)
    {
        for(int j=0;j<3;j++)
        {
            for(int k=0;k<3;k++)
            {
                cout<<a[i][j][k]<<' '; //打印数据,中间用“ ”分隔
            }
            cout<<endl;
        }
        cout<<endl;
    }
}
```

由于大量占有内存的关系,三维或更多维数组较少使用。如前所述,当数组定义之后,所有的数组元素都将分配到地址空间。例如,大小为(10, 6, 9, 4)的四维字符数组需要 $10 \times 6 \times 9 \times 4$ 即2160字节。如果上面的数组是两字节整型的,则需要4320字节;若该数组是双字型的(假定每个双字为8字节),则需要34560字节。也就是说多维数组的存储量随着维数的增加呈指数增长。

关于多维数组,需要注意一点:计算机要花大量时间计算数组下标,这意味着存取多维数组中的元素要比存取一维数组的元素花更多的时间。由于这些和其他原因,大量的多维数组一般采用C语言动态分配函数及指针的方法,每次对数组的一部分动态地分配存储空间。多维数组传递给函数时,除第一维外,其他各维都必须说明。例如,将数组m定义成:

```
int m[4][3][6][5];
```

那么接收m的函数应写成:

```
func1 (d)
```

```
int d[][3][6][5];
```

当然，如果愿意，也可加上第一维的说明。

本章小结

本章主要讲述了数组的应用技巧，通过排序算法展现了数组的各种不同定义和使用方式，同时也初步地讲述了有关算法与数据结构的知识。

习 题

1. 将一个二维数组进行转秩。
2. 将 N (N 为奇数) 行 N 列的表格，填充为从 1 到 $N*N$ 的数据，使得每行、每列和对角线的数字相加，都是相等的。
3. 将 N 行 N 列的表格，填充为由外向内且数字逐圈减少的数据。



第 8 章 函数与程序结构

本章内容：

- 顺序程序结构。
- 分支程序。
- 循环程序。
- 内联和重载。

重点：

函数的定义和使用。

目的：

掌握结构化程序设计技巧，利用函数分割计算逻辑代码，利用多文件程序设计方式组织大型程序的结构。

本书的前七章属于语言的基本知识讲解，因此在讲解过程中力求精确，内容是手册性质的，读者需要牢牢记住对应的知识点。从本章开始，每章的核心内容都为解决某一个或某一类问题而设置，因此在学习之前，先要了解需要解决的问题，再深入研究对应的知识点。本章以后，每章都有综合应用举例小节，在应用举例小节中使用的例题，会在后续的章节中继续讨论并完善程序。为剖析 C++ 更多的核心知识点，从第 8 章到第 14 章采用以下两个主线例子。

- 制作一个栈，设置该例子的目的是讲解对应知识点。
- 制作一个简单的游戏，设置该例子的目的是讲解如何举一反三，灵活运用知识点。

C++ 程序是由一系列函数构成的，函数是程序的基本逻辑组成单元。在本章中将向大家详细讲解函数的定义、使用规则和相关注意事项。

8.1 函数的概念

在程序中，对于同一任务可能在程序执行过程中要反复的进行，为每次执行都编写一段代码不仅会使代码长度明显增加，加大工作量，而且由于代码的冗长，会使程序出错的

几率增加，也增加了调试和查错的难度。为此，C++提供函数机制解决该问题。

把相关的语句组织在一起，并给它们注明相应的名称，利用这种方法把程序分块，这种形式的组合就称为函数(function)，通常也称为例程或过程(procedure)。函数是一个独立完成某项功能的语句块，根据用户的不同输入通过各种操作得到相应的输出。图 8-1 为函数的示意。

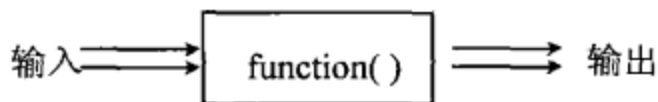


图 8-1 函数示意

有了函数以后，在编写代码时对于相同的功能只要编写一个通用的函数，在需要执行该功能时调用该函数即可。这使得代码的长度有了极大的缩减，当程序需要修改时只要修改一个函数即可，也使程序的修改与排错更加容易。

函数的分类有很多种，从应用角度分：函数可以分为库函数(标准函数)和用户自定义函数。其中库函数是由系统提供的基本函数，这些函数在多数程序中都会用到。用户自定义函数由程序员根据具体的项目需求进行编写，解决用户专门的需要。

从函数形式分，函数可以分为无参函数和有参函数，无参函数不接受外部参数，有参函数接受外部参数。

8.2 函数定义

同变量一样，C++的函数遵循先定义后使用的原则。要让函数能执行特定的功能，必须先给出函数的定义。定义函数需要指定函数的名称、参数类型和返回值类型。格式如下：

返回值类型标识符 函数名(形式参数表)；

返回值类型标识符指定函数的返回值类型，若函数不需要返回值，可将返回值类型设为 void。设置为 void 类型返回值的意思是，函数没有返回值或返回值没有任何意义。

函数名是函数的名称，是在程序中调用该函数的标识，函数名的命名规则遵循 C++关于标识符的命名规则。考虑到程序的可读性，以见名知意为宜。

第二个函数的声明中没有返回值，根据 C 语法的規定，默认的返回值类型为 int 型。而大部分 C++的编译器不支持这么做，因此在 C++中声明函数时要明确声明函数的返回值^①。

① 也有特例存在，在后续章节中讲述的“类”的“构造”和“析构”函数，就明确规定不能有任何返回类型的修饰，因此要牢记“凡事都有例外”。

函数的参数是可选项，当不需要向函数传递参数时，可将参数设为 `void` 或直接省略。因此，`void main()`和 `void main(void)`是等效的。

形式参数表用来指定需要向函数传递的参数类型，其格式如下：

```
<type1> name1, <type2> name2, ..., <type N> name N
```

例如：

```
void func(int x, int y)
{
    ...
}
max(int a, int b)
{
    ...
}
char * getname(int index)
{
    ...
}
```

当程序中需要使用到 `func` 函数的功能时，可以这样写：

```
void main()
{
    ...
    func(20, 30);
}
```

不管程序有多少个函数，程序总是从 `main` 函数开始运行(正是这个原因，所以才有规定，一个项目中有且只能有一个 `main` 函数)。当程序运行到 `func(20,30)`的时候，系统将 20 赋值给 `x`，30 赋值给 `y`，然后开始执行 `func` 函数内的代码(这就是函数调用)。当在 `func` 函数内遇到 `return` 语句，或执行到函数块的结尾时，程序返回到 `main` 函数中调用 `func` 函数的位置，接着执行该语句的下一条。

当函数被调用时，20 和 30 这两个数据是实际需要运算的原始数据，称作“实参”；而 `func` 函数形式参数表中的 `x`，`y` 仅仅是形式上参与运算的数据，它们的实际值是通过调用该函数的语句给的(20，30 这两个数)，因此它们称作“形参”。如果程序没有使用指针^①，则形参和实参之间的关系是：函数调用的时候，实参的值赋值给形参，之后实参和形参就没有联系了。

① 关于指针，将在第 9 章中进行详细的描述，只要记得这是不用指针的情况。

调用函数的时候,实参可以是常量^①,也可以是变量,但是有一个条件必须满足,就是形参的数据类型一定要和实参一致(或者可以进行隐式转换^②)。

```
int max(int a,int b)
//函数体
{
    if(a>b)
        return a;
    else
        return b;
}
```

上边该函数其实包括了两个部分:函数定义和函数体。`int max(int a,int b)`就是函数的定义,而其下的代码就是函数体,函数体具体描述了函数执行的工作。

8.3 函数声明

函数的使用,要遵守先定义后使用的原则,但是在特定的情况下,会出现难以解决的问题,例如下面的例子。

```
void func_a()
{
    ...
    if(...)
    {
        func_b();
    }
}
void func_b()
{
    ...
    if(...)
    {
        func_a();
    }
}
void main()
{
```

① 如果函数的形参是引用,则这里必须是变量,而不能是常量,请参考第9章“指针与引用”。

② 如果有函数重载,则必须完全一致,关于重载,请参考本章后续小节。

```
    func_a();  
}
```

在这个例子中,函数 `func_a` 在满足一定条件的情况下,调用函数 `func_b`;而函数 `func_b` 在满足一定条件的情况下,调用函数 `func_a`(这里一定要注意,是有条件的,而不是无条件的,如果是无条件的,则程序就是错的)。根据 8.1 节的学习可知,函数必须先定义后使用,因此理论上说:函数 `func_a` 调用了函数 `func_b`,那么函数 `func_b` 应该写在前面,但是 `func_b` 又调用了函数 `func_a`,因此 `func_a` 看起来又需要写在前面。这个问题不可调和,该怎么解决呢?这时,函数的声明就显得非常重要了。

所谓“函数声明”,就是用如下的形式来声明函数外,但是不书写函数体(请注意行末的分号)。

返回值类型标识符 函数名(形式参数表);

函数声明没有函数体,因此不存在哪个在前面的问题,当调用函数的时候,只要在此之前被声明过的函数,编译器都是认可的(必须在代码的其他部分有对应的函数体,否则最后还是要出错的)。这样上例的问题就可以这样来解决:在最前面增加这样两句函数的声明:

```
void func_a();  
void func_b();
```

这样不管是调用 `func_a` 还是 `func_b` 函数,编译器都已经识别了函数,编译即可正常进行。函数定义中声明的作用是让程序通过索引调用函数时能够顺利找到函数的入口地址。

在定义函数之前要先对函数进行声明,也可以在定义的同时声明函数。

例如:

```
void setname(int index, char & name)  
{  
    ...  
}  
void main()  
{  
    ...  
}  
double sqrt(double, double)  
{  
    ...  
}
```

这些都是正确的函数声明。

在函数声明的时候,参数名也不是必需的,函数声明只是用来告诉编译器函数的名字

和参数的类型。因此，函数声明可以写成：

```
void movi(int, int);
max(int, int);
char * getname(int);
void setname(int, char &);
```

由于函数声明中的形式参数只是为了增强程序的可读性而加入的，因此在定义时参数名可以不与前面声明时使用的参数名相同，但是声明和定义中的参数类型必须相同，否则C++的语法会认为这是两个完全不同的函数。这点在后面的函数重载部分会详细讲到。例如：

```
int getlevel( char * id);           //声明
...
int getlevel( char * playerid)      //定义
{
    ...
    return level;
}
```

8.4 函数调用

函数调用是通过使用参数调用函数来使函数完成特定的运算功能，函数的调用方式为：

函数名(实参列表)

其中，函数名即为需要调用的函数的名称，实参可以是常量、变量或表达式，例如：

```
sum(a, 2+b);
```

当调用参数为表达式时，函数在被真正执行前，程序会先对表达式求值，然后利用该结果作为参数调用函数完成计算。

函数的调用既可以作为程序中单独的一条语句，也可以放在表达式中。例如：

```
# include <iostream>
using namespace std;
double volume (double , double);      //函数原型
void main()
{
    double vol, r, h ;
    cin >> r >> h;
    vol = volume (r, h);
    cout << "Volume = " << vol << endl;
}
```

```
double volume (double radius, double height)
{
    return 3.14 * radius * radius * height;
}
```

本例中,函数调用发生在赋值语句(vol=volume(r,h);)中,函数的返回值被赋予变量 vol,例如:

```
#include <iostream>
using namespace std;

int a, b = 1;          //全程序可见
void main ()
{
    void f1();
    void f2();          //函数原型
    cout << "a1 = " << a << ", b1 = " << b << endl;
    f1();
    f2();
}
int i, j = 1;          //f1、f2 可见
void f1 ()
{
    cout << "a2 = " << ++a << ", b2 = " << ++b << endl;
    return;
}
void f2 ()
{
    cout << "a3 = " << ++a << ", b3 = " << ++b << endl;
    return;
}
```

在上例中,程序不需要函数的返回值,只要函数完成特定的功能,此时函数调用作为独立的语句被执行。

在调用函数时,需要注意以下几点。

- 当函数无参数时,参数列表可以为空,但()不能省略。
- 实参列表中多个参数用逗号间隔。
- 实参与形参个数一致;实参与形参类型一致;参数从左向右依次匹配,当指定的参数个数少于函数声明中的参数个数时,未指定的参数使用默认参数。若没有默认参数或参数类型不正确时,程序编译会报错。
- 实参变量对形参变量的数据传递是“值传递”,即单向传递,只由实参传给形参,而不能由形参传回给实参。

8.5 变量的作用域类型

“语言的作用域规则”是一组确定一部分代码是否“可见”或可访问另一部分代码和数据的规则。

C++语言中的每一个函数都是一个独立的代码块。一个函数的代码块是隐藏于函数内部的，不能被任何其他函数中的任何语句(除调用它的语句之外)所访问。例如，用 `goto` 语句跳转到另一个函数内部是不可能的。构成一个函数体的代码对程序的其他部分来说是隐蔽的，它既不能影响程序其他部分，也不受其他部分的影响。换言之，由于两个函数有不同的作用域，定义在一个函数内部的代码数据无法与定义在另一个函数内部的代码和数据相互作用。

C++语言中所有的函数都处于同一作用域级别上。这就是说，把一个函数定义于另一个函数内部是不可能的。而变量由于其不同的定义位置(函数内部或是所有函数之外)，因而拥有了不同的作用域。

8.5.1 局部变量

在函数内部定义的变量称为局部变量。局部变量仅可由其被定义的模块内部的语句所访问。换言之，局部变量在自己的代码模块之外是不可知的，它们仅存在于被定义的当前执行代码块中，这种模块范围也称作作用域。即局部变量在进入作用域时生成，在退出作用域时消亡。作用域以左花括号开始，以右花括号结束。

定义局部变量最常见的代码块是函数。例如，考虑下面两个函数。

```
func1()
{
    int x;
    x = 10;
}
func2()
{
    int x;
    x = -19;
}
```

整数变量 `x` 被定义了两次，一次在 `func1()` 中，一次在 `func2()` 中。`func1()` 和 `func2()` 中的 `x` 互不相关，其原因是每个 `x` 作为局部变量仅在被定义的块内具有可见性，因此不会出现变量重复定义。

学过 C 语言的读者可能都记得函数中的局部变量都要在函数体顶部定义。例如：

```
void f()
{
    int t;
    char s[80];           //数组定义
    scanf("%d",t);
    if(t==1)
    {
        printf("enter name:\n");
        gets(s);          //输入字符串
        process(s);        //功能函数调用
    }
    return ;
}
```

`char s[80]`定义了一个可存放 80 个字符类型数据的数组。但是，数组 `s` 也许根本就用不到，但还是必须在函数的最开始就声明它。即在函数开始就占据了 80 字节的空间而没有使用，且直到函数调用结束才被释放。现在，在 C++ 中这种情况已经不存在了，C++ 允许在程序的任意一点定义变量(包括全局变量)。例如：

```
void f()
{
    int t;
    cin>>t;
    if(t==1)
    {
        char s[80];       //此变量仅在这个块中起作用
        cout<<"enter name:"<<endl;
        gets(s);          //输入字符串
        process(s);        //函数调用
    }
}
```

这里的局部变量 `s` 就是在 `if` 块入口处才建立，相应的作用域也由原来的整个函数变为只在 `if` 语句的内部有效，在其出口处消亡。因此 `s` 仅在 `if` 块中可见，而在其他地方均不可访问，甚至在包含它的函数内部的其他部分也不行。

在一个条件块内定义局部变量的主要优点是仅在需要时才为之分配内存。这是因为局部变量仅在控制进行到它们被定义的块内时才进入生存期。虽然在大多数情况下这并不十分重要，但对于为了提高效率而不惜牺牲空间的游戏程序设计，这就变得十分重要了。

由于局部变量随着它们被定义的模块的进出口而建立或释放，它们存储的信息在块工作结束后也就丢失了。当访问一函数时，它的局部变量被建立，当函数返回时，局部变量

被销毁。局部变量的值不能在两次调用之间保持。

8.5.2 全局变量

与局部变量不同，全局变量贯穿整个程序，并且可被任何一个模块使用。它们在整个程序执行期间保持有效。全局变量定义在所有函数之外，可由函数内的任何表达式访问。

在下面的程序中可以看到，变量 `count` 定义在所有函数之外、函数 `main()` 之前。但其实它可以放置在第一次被使用之前的任何地方，只要不在函数域内就可以。实践表明，定义全局变量的最佳位置是在程序的顶部。定义在其他位置的全局变量，对于出现在它之前的代码具有不可见性，即对于全局变量(或是局部变量)，都只对出现在它后面的代码有效。

当在函数内部定义的局部变量与全局变量重名时，在该函数内部全局变量不可见。例如：

```
int count;                                //count 是全局变量
void main()
{
    count = 100;
    func1();
}
func1()
{
    int temp;
    temp = count;
    func2();
    printf("count is %d",count);    //打印 100
}
func2()
{
    int count;
    for(count = 1; count < 10; count++)
        putchar('.');                //打印出"。"
}
```

仔细研究此程序后，可见全局变量 `count` 既不是 `main()` 也不是 `func1()` 函数定义的，但两者都可以使用它。函数 `func2()` 也定义了一个局部变量 `count`。当 `func2` 访问 `count` 时，它仅访问自己定义的局部变量 `count`，而不是全局变量 `count`。切记，全局变量和某一函数的局部变量同名时，该函数内直接对该名的所有访问仅针对局部变量，对全局变量无影响。如果忘记了这点，即使程序看起来是正确的，也可能导致运行时的奇异行为。

全局变量由 C++ 编译程序在动态存储区之外的固定存储区域中存储。当程序中多个函数都使用同一数据时，全局变量将是很有效的。然而，由于下列三种原因，应避免使用不

必要的全局变量。

- 不论是否需要，它们在整个程序执行期间均占有存储空间。
- 由于全局变量必须依靠外部定义，所以如果在使用局部变量就可以达到其功能时使用了全局变量，将降低函数的通用性，这是因为它要依赖其本身之外的东西。
- 大量使用全局变量时，不可知的和不需要的副作用将可能导致程序错误。如在编制大型程序时有一个重要的问题：变量值都有可能在程序其他地点偶然改变。

结构化语言的原则之一是代码和数据的分离。C++语言是通过局部变量和函数的使用来实现这一分离的。下面用两种方法编制计算两个整数乘积的简单函数 `mul()`。

通用的	专用的
<code>mul(x,y)</code>	<code>int x,y;</code>
<code>int x,y;</code>	<code>mul()</code>
<code>{</code>	<code>{</code>
<code>return (x*y);</code>	<code>return (x*y);</code>
<code>}</code>	<code>}</code>

两个函数都是返回变量 `x` 和 `y` 的积，但是通用的(或称为参数化版本)可用于任意两整数之积，而专用的版本仅能计算全局变量 `x` 和 `y` 的乘积。

8.6 变量的存储类型

8.6.1 动态存储变量

从变量的作用域原则出发，可以将变量分为全局变量和局部变量；从变量的生存期来分，可将变量分为动态存储变量及静态存储变量。

动态存储变量可以是函数的形式参数、局部变量、函数调用时的现场保护和返回地址。这些动态存储变量在函数调用时分配存储空间，函数结束时释放存储空间。有时为了提高速度，还可以将局部的动态存储变量定义为寄存器型的变量，定义的形式为在变量的前面加关键字“`register`”。例如：

```
register int x,y,z;
```

这样的好处是：变量的值不是存入内存，而是保存在 CPU 的寄存器中，以使速度大大提高。但是 CPU 内的寄存器数量有限，对这种变量的个数是要严格控制的。

8.6.2 静态存储变量

在编译时分配存储空间的变量称为静态存储变量，其定义形式为在变量定义的前面加上关键字“static”。例如：

```
static int a=8;
```

C++中定义的静态存储变量无论是做全局变量或是局部变量，其定义和初始化均在程序编译时自动进行，编译器自动将静态存储类型变量的值清为零。而在调用函数结束时，作为全局变量，静态存储变量不消失并且保留原值。例如：

```
void main()
{
    int f();           //函数声明
    int j;
    for(j=0; j<3; j++)
        cout<<"f() "<<j+1<<" times is "<<f()<<endl;
}

int f()               //函数定义
{
    static int x=1;
    x++;
    return x;
}
```

程序运行结果如下：

```
f() 1 times is 2
f() 2 times is 3
f() 3 times is 4
```

从上述程序看，函数 f() 被调用三次，由于局部变量 x 是静态存储变量，它是在编译时分配存储空间，故每次调用函数 f() 时，变量 x 不再重新初始化，而是保留上次加 1 后的值，于是得到上面的输出。

8.7 函数返回值

通常在函数完成一项计算任务以后，需要向程序返回一个计算结果，称为函数的返回值。函数的返回值由 return 语句给出。return 语句的格式为：

```
return [返回值或表达式];
```

例如：

```
return 0;
return NULL;
return a>b?a:b;
```

若函数不需要返回值(即返回值为 `void` 类型), 在函数执行完时, 可以不必写 `return` 语句或写成 `return;` 或 `return NULL;`。在这种情况下, `return` 语句的功能是结束函数的执行并返回调用函数继续执行。

结束函数的方式有以下两种。

(1) 程序执行完函数的最后一条语句后自动返回。这是最简单的情形, 例如:

```
void maxmum(int x, int y, int z)
{
    int max;
    max=x>y?x:y;
    max=max>z?max:z;
    cout<<"The maxmum value of the 3 data is "<<max<<endl;
}
```

当函数执行完最后一条语句, 即在屏幕上显示完 `The maxmum value of the 3 data is max` 后自动返回。

(2) 调用 `return` 语句强制函数返回。当程序中存在多种分支, 对每个分支执行完相应的操作后即完成函数的功能需要返回主调函数时, 常用这种方式。例如:

```
int max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
```

在本例中, 当判断出 `a>b` 后, 函数只需向调用函数返回一个最大值 `a` 即可, 无须再继续向下执行, 因而使用 `return` 语句, 强制函数返回。

需要说明的是, 在编程时由于分支数目繁多, 可能会丢掉某些分支, 造成函数没有返回值的情况。例如:

```
int compare( int a, int b)
{
    if(a>b)
        return 1;
    else if(a<b)
        return -1;
}
```


在上例中，若 $a=b$ ，则函数无返回值与之对应。应将函数改为：

```
int compare( int a, int b)
{
    if(a>b)
        return 1;
    else if(a<b)
        return -1;
    else
        return 0;
}
```

建议在含有多路返回分支的函数中，在最后都单加一个返回语句以保持程序的健壮性。

函数返回值的类型最好和函数声明的返回值类型一致，否则在函数返回时将发生类型转换。这种类型转换有可能如下：

```
int calarea(double radus)
{
    return 3.14*radus*radus;
}
```

在本例中，求出的圆面积为 `double` 类型，而函数声明时指定的返回值类型为 `int` 型，在返回时系统自动将会把 `double` 转换为 `int` 型。当自动类型转换无法完成时，需要进行手工的强制类型转换。

8.8 默认函数参数

原则上来说，C++对函数参数的个数没有限制，但是过多的参数会对函数的使用产生负面影响。过多的参数不仅会使程序员难以记住参数的个数和类型，增加程序员的负担，而且由此带来的调试任务也会急剧增加。因此，建议函数的参数尽量不要超过 8 个，当必须向函数传递多个参数时，可将多个参数合并为一个类或是结构体^①，以简化参数的数目。

对于大多数程序来说，尤其在游戏中，许多函数的调用参数并不是随机的，有很多情况下函数的调用参数完全相同。举个最简单的例子：当一个新玩家开设了一个游戏账号时，游戏程序需要为该玩家创建一个 `player` 对象。对于这个新创建的对象，除了名字和种类可能稍有不同外，多数属性应该是相同的，否则会严重破坏游戏的均衡性。想一想，同样是开设一个游戏账号，玩家的 `player` 刚开始玩时被设置为 0 级，而别人的刚开始玩时被设置

① 类和结构体，在后续章节中进行说明。

为 100 级，玩家会同意吗？当然不可能。因此，最简单的办法便是为每个玩家在开始时都设置相同的基本属性。

那么，程序中如何做到这点呢？对于传统的 C 程序或是 VB 程序来说，肯定是在每次创建人物时都使用一大堆繁琐的参数调用函数来实现。然而，C++设计了更有效的做法：默认参数。

C++函数在编写时可以预先为参数设置默认值，在程序调用该函数时，若用传入的参数调用，函数将采用传入参数进行计算；若没有指定参数，函数可以使用默认参数进行计算。例如：

```
#include <iostream>
using namespace std;

int createplayer(char *name, int sex=1, int level=0, int life=100, int experience=0);

int main()
{
    createplayer("Player1");
    cout<<endl<<endl;
    createplayer("Player2",0,30);
    return 0;
}

int createplayer(char *name, int sex=1, int level=0, int life=100, int experience=0)
{
    cout<<"We're creating a player called "<<name<<endl;
    if(sex)
    {
        cout<<"He is a man"<<endl;
        cout<<"His level is "<<level<<endl;
        cout<<"His life is "<<life<<endl;
        cout<<"His experience is "<<experience<<endl;
    }
    else
    {
        cout<<"She is a girl"<<endl;
        cout<<"Her level is "<<level<<endl;
        cout<<"Her life is "<<life<<endl;
        cout<<"Her experience is "<<experience<<endl;
    }
    cout<<"Now, the player has been created.You can enjoy your game then."
        <<"May you good luck!"<<endl;
    return 1;
}
```

程序运行结果如下:

```
We're creating a player called Player1
He is a man
His level is 0
His life is 100
His experience is 0
Now, the player has been created. You can enjoy your game then. May you good luck!
```

```
We're creating a player called Player2
She is a girl
Her level is 30
Her life is 100
Her experience is 0
Now, the player has been created. You can enjoy your game then. May you good luck!
```

在本例中,第一次调用函数 `createplayer()` 时只给出了一个参数:名字“`player`”,其他自动取默认值,创建了一个男性游戏者;第二个函数的前几个参数由程序员进行了指定,在函数运行时,不再使用默认的参数,而是使用指定的参数值计算,而对于后面未指定的参数依然使用默认值。

默认形参值必须从右向左顺序声明,并且在默认形参值的右面不能再有非缺省形参值的参数。这是因为在函数调用时,实参取代形参是按照从左向右的顺序。例如:

```
int add(int x,int y=5,int z=6);           //正确
int add(int x=1,int y=5,int z);           //错误
int add(int x=1,int y,int z=6);           //错误
```

相反的,以指定的参数调用带有默认参数的函数时,参数的指定顺序为从左向右顺次指定,不允许间隔指定,对于没有默认值的参数必须指定参数。例如,对于函数:

```
int add(int x,int y=5,int z=6);
```

调用时:

```
add(10,20,30);           //正确
add(10,20);               //正确
add(10);                  //正确
add(10,,30);              //不正确,不允许跨参数指定
add();                    //不正确,未对 x 指定参数
```

在调用带有默认参数的函数时,若调用出现在函数体实现之前,则默认形参值必须在函数原形中给出;若调用出现在函数体实现之后,默认形参值需在函数实现时给出;并且默认函数参数的声明只可出现一次。例如:

```
//调用在实现前
int add(int x=5,int y=6);
void main(void)
{
    add();
}
int add(int x,int y)
{
    return x+y;
}
```

```
//调用在实现后
int add(int x=5,int y=6)
{
    return x+y;
}
void main(void)
{
    add();
}
```

在相同的作用域内，默认形参值的说明应保持唯一，但如果不同的作用域内，允许说明不同的默认形参。例如：

```
int add(int x=1,int y=2);
void main(void)
{
    int add(int x=3,int y=4);
    add();          //使用局部默认形参值：3+4
    fun();
}
void fun(void)
{
    ...
    add();          //使用全局默认形参值：1+2
}
```

程序运行结果为：7，3。

8.9 内联函数

函数的引入使得编程者只关心函数的功能和使用方法，而不必关心函数功能的具体实现；函数的引入可以减少程序的目标代码，实现程序代码和数据的共享。但是，函数调用也会带来降低效率的问题，因为函数调用实际上是将程序执行顺序转移到函数所存放的内

存中的某个地址，将函数的程序内容执行完后，再返回到转去执行该函数前的地方。这种转移操作要求在转去前要保护现场并记忆当前执行的地址，转回后先要恢复现场，并按原来的保存地址继续执行。因此，函数调用要有一定的时间和空间方面的开销，这将影响其效率。特别是对于一些函数体代码不是很大，但又频繁地被调用的函数来讲，调用开销的效率问题更为重要。引入内联函数实际上就是为了解决这一问题。

在程序编译时，编译器并不对内联函数进行调用，而是将程序中出现的内联函数的调用表达式用内联函数的函数体来进行直接替换。显然，这种做法不会产生转去转回的问题，但是由于在编译时函数体中的代码被替代到程序中，因此会增加目标程序代码量，进而增加空间开销，而在时间开销上不像函数调用时那么大，可见它是以目标代码的增加为代价来换取时间的节省。

定义内联函数的方法很简单，只要在函数定义的头前加上关键字 `inline` 即可。内联函数的定义方法与一般函数一样，其形式如下。

```
inline 返回值类型 函数名(形式参数表);
```

如：

```
inline int add(int x, int y, int z)
{
    return x+y+z;
}
```

在程序中调用其函数时，该函数在编译时被替代，而不是像一般函数那样是在运行时被调用。

内联函数具有一般函数的特性，它与一般函数的不同之处在于函数调用的处理。一般函数进行调用时，要将程序执行权转到被调用函数中，然后再返回到调用它的函数中；而内联函数在调用时，是将调用表达式用内联函数体来替换。在使用内联函数时，应注意如下几点。

- 在内联函数内不允许用循环语句和 `switch` 语句。有这些语句的存在，即使是加上 `inline` 关键字编译器也不会把它当作内联函数。递归函数也不可以作为 `inline` 函数。

`inline` 指示对编译器来说只是一个建议，编译器可以选择忽略该建议。因为把一个函数声明为 `inline` 函数并不见得它真的适合在调用点上替换展开。一般的，`inline` 机制用来优化小的、只有几行的、经常被调用的函数。

- 内联函数的定义必须出现在内联函数第一次被调用之前。`inline` 函数对编译器而言必须是可见的，以便它能够在调用点展开该函数。因为这个原因，与非 `inline` 函数不同的是 `inline` 函数必须在调用该函数的每个文本文件中定义。因此，建

议把 `inline` 函数的定义放到头文件中，在每个调用该 `inline` 函数的文件中包含该头文件。

- 类中所有在类说明内部定义的函数是内联函数。(详见类部分)

8.10 函数重载

在以往的程序设计语言中，函数是不允许重名的。对于一个求绝对值的函数，由于参数类型的不同，需要为每种参数单独写一个不同名的函数。例如：

```
int    abs_int(int);
float  abs_float(float);
double abs_double(double);
long   abs_long(long);
```

而在调用函数时，针对每种不同的参数又都要记住相应函数的函数名。例如：

求整数 4 的绝对值，需要调用 `abs_int(4)`；

求长整型数-30000 的绝对值，需要调用 `abs_long(4)`；

求浮点数 27.65 的绝对值，需要调用 `abs_float(27.65f)`；

程序员需要把大量的精力放在了记忆这些函数名和参数类型上，给函数的使用带来了极大的不便。

同时，由于在调用函数时系统可以进行自动类型转换，这样即使在某些场合使用了错误的函数，程序执行的可能是完全错误的操作，但编译器却无法在语法上判断出来，这给程序带来隐患。C++提供了函数重载机制来帮助用户降低这种风险。

所谓函数的重载，就是两个或两个以上的 C++函数可用同一个函数名。编译器将根据函数调用时提供的实际参数的类型和个数与形式参数匹配来决定调用哪个函数。函数重载的原则是：重载函数应当执行相同的功能，它们的差异在于参数不同。让重载函数执行不同的功能，不是良好的编程风格。

如上面的几个求绝对值的函数就可以声明为：

```
int abs(int);
float abs(float);
double abs(double);
long abs(long);
```

虽然这里也同样要为 4 个不同的函数编写代码，但是在调用函数时，无论对于哪种类型的参数，调用方式均为 `abs(实参)`，例如：

```

void main()
{
    int i;
    long j=14473;
    double k=37.22;
    abs(i);           //int abs(int);
    abs(j);           //long abs(long);
    abs(k);           //double abs(double);
}

```

程序到底调用哪个函数的工作由编译器根据调用参数的类型自行判断。

下面再看一个具体的函数重载的例子。

例如：编写两个名为 `add` 的重载函数，分别实现两整数相加、两实数相加的功能。

```

#include<iostream>
using namespace std;

void main(void)
{
    int m, n;
    double x, y;
    //内联函数声明
    int add(int m, int n);
    double add(double x, double y);

    cout<<"输入两个整数: ";
    cin>>m>>n;
    cout<<"整数计算结果 "<<m<<'+'<<n<<"="<<add(m,n)<<endl;

    cout<<"输入两个浮点数: ";
    cin>>x>>y;
    cout<<"浮点数的计算结果 "<<x<<'+'<<y<<"="<<add(x,y)<<endl;
}

int add(int m, int n)
{
    return m+n;
}

double add(double x, double y)
{
    return x+y;
}

```

程序运行结果为:

输入两个整数: 3 5

整数的计算结果 $3+5=8$

输入两个浮点整数: 2.3 5.8

浮点数的计算结果 $2.3+5.8=8.1$

函数重载是有一定条件的: 重载函数至少要在参数个数、参数类型或参数顺序上有所不同。如果几个函数的函数名称相同, 但参数类型、参数个数、参数顺序有所不同, 则认为是合法的重载函数。例如:

```
void f1();  
void f1(int);  
void f1(int ,double);  
void f1(char);
```

以上这些都是正确的函数重载的例子。

但如果仅仅是返回类型不同, 则不是合法的重载函数。因为编译器在编译时, 需要根据函数参数的类型判断到底调用哪个函数。若只是返回值不同, 则编译器将无法作出正确的识别。例如:

```
float sqrt(float);  
double sqrt(float);  
void fun(float fdata1, float fdata2)  
{  
    float f;  
    double d;  
    f = sqrt(fdata1);  
    d = sqrt(fdata2);  
}
```

在例子中的两次函数调用都无法确定该调用哪个函数。



注意: ① 在对函数进行重载时应避免因函数带有默认参数而产生的二义性问题。例如:

```
void func(int a);  
void func(int a, int b=2);  
void main()  
{  
    int a;  
    a = 3;  
    func(3);  
}
```

在本例中, 对函数 `func()` 的调用将产生严重的二义性。由于第二个函数存在默

认参数,使得两个函数都可以非常完美地匹配函数的调用方式,编译器在编译时根本无法知道该调用哪个函数。

② 重载函数过少依然会产生二义性问题。例如:

```
void func(long a);
void func(double a);
void main()
{
    int a;
    a = 3;
    func(3);
}
```

由于无法找到参数类型为 int 的函数,即无法找到精确匹配,则系统会自动类型转换,但从 int 既可以转换为 long 型也可以转换为 double 型,编译器依然无法确定调用哪函数。

因此,这里给出的建议是:在函数重载时,尽可能把所有可能用到的重载情况全部写出,以避免因类型转换带来的影响。

8.11 作用域

作用域是标识符在程序中有效的范围,标识符的引入与声明有关,作用域开始于标识符的声明处。

C++的作用域范围分为:局部作用域(块作用域)、函数作用域、函数原型作用域、类和文件作用域。

8.11.1 局部作用域

当标识符的声明出现在用一对花括号所括起来的一段程序(块、复合语句)内时,该标识符的作用域从声明点开始,到块结束处为止,该作用域的范围具有局部性,称此作用域为局部作用域(块作用域)。例如:下面的代码描述了局部作用域。

```
void fn ( int y )           //y 的作用域从此开始
{
    int x=1;                //x 的作用域从此开始
    if (x>y)
        cout << x << endl;
    else
```

```

        cout << y << endl;
    //...
} //x 和 y 的作用域到此结束

```

变量在语句块作用域中，例如：

```

void fn()
{
    if(int i=5) //i 的作用域从此开始
        i=2*i;
    else
        i=100;
    //i 的作用域到此结束

    cout<< i <<endl;
}

```

8.11.2 函数作用域

函数作用域在整个函数内都起作用。唯一拥有函数作用域的标识符是标号，函数内的标号可在函数内的任何位置被 goto 语句使用。例如：

```

void main()
{
    int sum=0,i=1;
bg:if(i<=10)
{
    sum+=i++;
    if(sum>30)
        goto end;
    goto bg;
}
end:cout<<i<<' '<<sum<<endl;
}

```

输出的结果为：

```
9 36
```

这里 bg 和 end 两个标号在函数 main 中的任何位置都可以被引用。

特别容易使人迷惑的是，局部变量拥有的是局部作用域而不是函数作用域。因为局部变量是从其定义位置开始起作用，只在块内或局部可见，并且在嵌套块的内部还可定义同名局部变量。因此，局部变量不具有函数作用域。

8.11.3 函数原型作用域

函数原型作用域是 C++ 程序中最小的作用域，它是指在函数原型声明时形式参数的作用范围，开始于函数原型声明的左括号，结束于函数原型声明的右括号。例如，设有下列原型声明：

```
double Area(double radius);
```

则 `radius` 的作用域仅在括号内部，不能用于程序正文的其他地方。也正是因为 `radius` 的作用域仅在括号内部，所以在声明函数原形时形参可以省略，声明中加上形参的目的是提高程序的可读性。

除了上述几种作用域外，还有文件作用域，相关知识点请参考本章的 8.14.3 节。

8.12 可见性与生命期

8.12.1 可见性

可见性从另一角度表现标识符的有效性。标识符在某个位置可见，表示该标识符可以被引用。可见性与作用域是一致的。作用域指的是标识符有效的范围，而可见性是分析在某一位置标识符的有效性。

可见性在分析两个同名标识符作用域嵌套的特殊情况时，非常有用。在内层作用域中，外层作用域中声明的同名标识符是不可见的。当在内层作用域中引用这个标识符时，表示的是对内层作用域中声明的标识符的引用。

例如，下面的程序定义了 3 个不同作用域的同名变量，在访问它们时，可见性起了作用。

```
int id=3;
void main()
{
    int id=5;
    {
        int id;
        id=7;
        cout <<"id=" <<id <<endl;    //输出 7
    }
    cout <<"id=" <<id <<endl;        //输出 5
}
```

程序运行结果如下：

```
id=7  
id=5
```

上面程序里最外层的全局变量 `id` 有文件作用域；内层的局部变量 `id` 有块作用域。内层的变量 `id` 隐藏了外层的变量 `id`，因此，在主函数 `main()` 内，定义了局部变量 `id` 之后，就不能直接访问文件作用域的全局变量 `id` 了；最内层的变量 `id` 又隐藏了次外层的变量 `id`。

在最内层的块作用域结束后，次外层的块作用域又变得可见了，所以运行结果的第二行输出 `id` 的值为 5。

标识符的可见性范围不超过作用域，作用域则包含可见范围。例如，下面的代码进一步说明了可见性与作用域的关系。

```
{  
    int i;  
    char ch; =3;  
    {  
        double i;  
        i=3.0e3;           //int i 被隐藏  
        ch='A';           //char ch 仍可见  
    } //double i 的作用域结束  
    i+=1;                 //int i 可见  
} //int i, char ch 作用域结束
```

该代码的图示如图 8-2 所示。

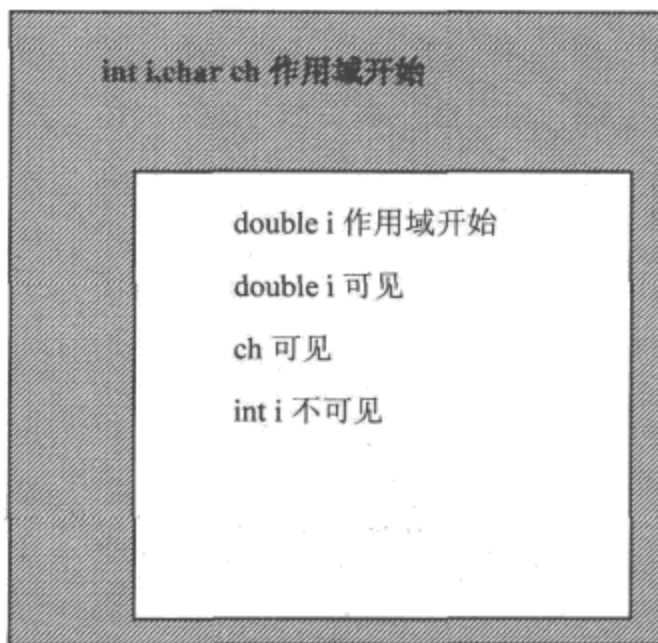


图 8-2 作用域与可见性

图中阴影部分为 `int i` 和 `char ch` 的作用域。进入内部块后，`double i` 遮住了 `int i`，使得 `int i` 不可见，但变量 `ch` 仍可见。所以在内部块中，`double i` 的作用域和可见性是一致的，`int i` 作用域存在，但不可见。在外部块中，`ch` 的作用域与可见性是一致的，因为 `ch` 的可见性渗透至内部块中。

如果被隐藏的是全局变量，则可用符号 “`::`” 来引用该全局变量。相关的内容可参见 11.3 节。

例如，下面的代码定义了同名的全局变量和局部变量，但仍可在局部作用域中访问全局变量。

```
int s=0;
void f()
{
    float s=3.0;
    int a;
    {
        float a=2.0;
        ::a=1;           //没有全局变量 a
        ::s=1;           //指全局变量 s
        s=2.0;           //指 float s
    }
}
```

8.12.2 生命期

生命期也叫生存期。生命期与存储区域密切相关，存储区域主要有代码区、数据区、栈区和堆区，对应的生命期为静态生命期、局部生命期和动态生命期。

1. 静态生命期

变量在固定的数据区中分配空间的，具有静态生命期，包括全局变量、静态全局变量和静态局部变量。具有文件作用域的变量具有静态生命期，这种生命期与程序的运行相同，只要程序一开始运行，这种生命期的变量就存在，当程序结束时，其生命期就结束。

说明：

- 静态生命期的变量，若无显式初始化，则自动初始化为 0。
- 函数驻在代码区也具有静态生命期。在函数内部可以声明静态生命期的变量，即静态局部变量(加 `static`)。

2. 局部生命期

在函数内部声明的变量或者在块中声明的变量具有局部生命期。这种变量的生命期开始于程序执行经过其声明点时，而结束于其作用域结束处。所以具有局部生命期的变量也具有局部作用域。但反之不然，具有局部作用域的变量若为局部变量，则具有局部生命期；若为静态局部变量，则具有静态生命期。

说明：

- 具有局部生命期的变量驻在内存的栈区。
- 具有局部生命期的变量如果未被初始化，则内容不可知。

3. 动态生命期

这种生命期由程序中特定的函数(`malloc()`和 `free()`)调用或由操作符(`new` 和 `delete`)来创建和释放。

具有这种生命期的变量驻在内存的堆中。当用 `malloc()`函数或 `new` 操作符为变量分配空间时，生命期开始；当用 `free()`函数或 `delete` 操作符释放该变量的空间或程序结束时，生命期结束。

8.12.3 补充说明

(1) 在同一个函数内，不能定义两个同名的变量。

(2) 在局部变量一节说明过：在两个不同的函数内，在一个函数内部定义的局部变量，如果在另外一个函数中也有相同的局部变量，两个变量之间没有任何联系，因此互相不影响。

(3) 如果定义了一个全局变量，而在某函数内又定义了同名的局部变量，则在该局部变量的有效范围内，任何有关该变量的访问，默认都是访问局部变量。

8.13 综合应用举例

首先思考一个问题：求任意长度的一个算术四则运算表达式的值。

这个问题有以下两个解决办法。

方法一：遍历所有的运算符号，找出最优先的表达式，计算结果，然后把结果放回表达式，重复这个过程，直到只有一个数据。很明显，这个方法效率很低。

方法二：利用第4章中提到的逆波兰式，先将中缀表达式转换为后缀表达式，之后再计算。这个方法，只需要遍历两次表达式，因此效率比第一个高。

根据上述描述，采用第二种方式无疑是好的。而在计算的过程中，需要使用到“栈”^①来临时存储数据。应用程序不应该使用系统栈，因此只能重新构造一个栈来使用，考虑到程序中需要存储数据以及大量操作栈中的数据，现做如下设计。

(1) 定义一个全局数组，用于做数据存储空间；同时定义一个全局的整数变量，初始化为0，用于表示目前数据空间中存储的数据个数，同时也做数据存取位置的指示器。例如当该变量为3时，表示空间中存储了3个数据，也就是说，数组中索引为3的元素，同时可以存储新的数据。如果想取数据，则取索引为2的元素所保存的数据。

(2) 定义两个函数，其中一个用于向空间中存数据，另外一个从空间中取数据。应用程序不直接访问全局数组，仅通过这两个函数来存取数据，这样就可以保证按照栈的原则访问数据。

(3) 代码设计如下：

```
int data[200];
int pos = 0;

void push(int a)
{
    data[pos++] = a;
}
int pop()
{
    return data[--pos];
}
```

将 data 和 pos 设计为全局变量，是因为在 push() 和 pop() 两个函数中都需要访问这些数据。函数的参数和返回值也是根据函数所做的事来决定的。

当程序需要存数据的时候，只需要调用 push 函数即可。函数首先存储数据到 pos 指定

① 所谓“栈”，就是一个存储数据的空间，只是这个空间有点特殊，后存储进的数据访问的时候优先，数据只能按顺序访问，不能随机访问。例如，一个内径大小刚好和乒乓球一样大的筒，其中一头是封死的，另外一头开口的。如果放进一个乒乓球，之后再放进一个乒乓球，此时，第二个乒乓球阻挡了第一个乒乓球的进出路线。当想取出乒乓球的时候，只能先取第二个，再取第一个，此时这个球筒就成了一个具有后进先出原则的容器。这时候，可以称该球筒是一个“栈”，所存储的数据是“乒乓球”。

的位置，然后将 pos 自增 1；而当取数据的时候，首先将 pos 自减 1，再将对应位置的数据返回，这样既存取了数据，又保证了 pos 变量的值一直保持和数据状态同步。只要应用程序不直接访问 pos 和 data 这两个全局变量，就可以保证对 data 的访问，满足了栈的原则。这样可以通过这个栈做数据临时存储空间，进行表达式的计算了。表达式的具体计算细节，请参考数据结构对应的章节，就不做深入研究，这里仅仅讨论栈的设计。

经过这个设计，栈是可以用了，但这个栈并不是完善的，有很多问题需要改进。例如当 data 中已经存储了 200 个数据的时候，再试图存数据将产生数组越界问题。此时可对存数函数做如下更改。

```
bool push(int a)
{
    If(pos==200)
    {
        return false;
    }
    data[pos++] = a;
    return true;
}
```

这样在每次使用存功能的时候，先判断函数的返回值是否为真，以进一步判断数据是否存储成功，可以保证数据访问不越界。

经过这些设计，这个栈可以简单使用了，但依旧很不完善。例如，这个栈只能存储整数，最多只能存储 200 个数据等。因此在后续的章节中，随着学习的深入，将进一步完善该程序。

8.14 递归函数

函数直接或间接地调用函数本身，称为递归调用。前者称为直接递归，后者称为间接递归。递归调用的函数称为递归函数。由于递归非常符合人们的思维习惯，而且许多数学函数及许多算法或数据结构都是递归定义的，因此，递归调用颇具实用价值。

8.14.1 递归函数举例

从一个实际的例子下手，分析一下递归程序是如何实现的。以求阶乘运算为例。之所以选择它，是因为阶乘($n!$)定义就是递归的。阶乘的定义为：


```
n!=1          //n=0 时
n!= n*(n-1)!  //n>1 时
```

按照普通的算法，将一个一个一个地乘出最后结果。但是，使用递归的方法会更为快速。递归程序如下。

```
#include <iostream>
using namespace std;
long factor(int n)
{
    if (n==0||n==1)
        return (1);
    else
        return (n*factor(n-1));
}

void main()
{
    int x;
    cout<<"请输入级数(小于10): ";      //请考虑为什么要小于10
    cin>> x;
    cout<<factor(x);
}
```

程序运行时，输入任意一个小于10的整数值，就可以得到它的阶乘值。

8.14.2 递归调用过程分析

根据变量的作用域和生命期，递归程序在执行过程中每次对自身的调用，其形参尽管变量名相同，但每一次从实参复制来的值是不同的，而且每次调用完成后，形参的值会自动压入堆栈，因此不会互相混淆，这是理解递归程序执行过程的基本依据。

递归程序的执行过程可分为两个阶段：回推和递推。

所谓“回推”，是指在求 $n!$ 时，必须先求出 $(n-1)!$ ，而要求 $(n-1)!$ ，必须先求出 $(n-2)!$ ，如此往回推下去，直至回推到底求 $1!$ 。由于 $1!$ 已经知道了，整个回推过程即告结束。接下来就是递推过程。所谓“递推”，实际上是回推的逆过程，即根据计算公式，由 $1!$ 求得 $2!$ ，再由 $2!$ 求得 $3!$ ，以至于最终求得 $n!$ 。假设要求 $5!$ ，程序的执行过程如图 8-3 所示。

从上面的执行过程可以看出，在“回推”阶段，形参的值依次压入堆栈；在“递推”阶段，形参的值再依次弹出，在弹出的同时参加递推公式的计算，最终求出所需要的结果。

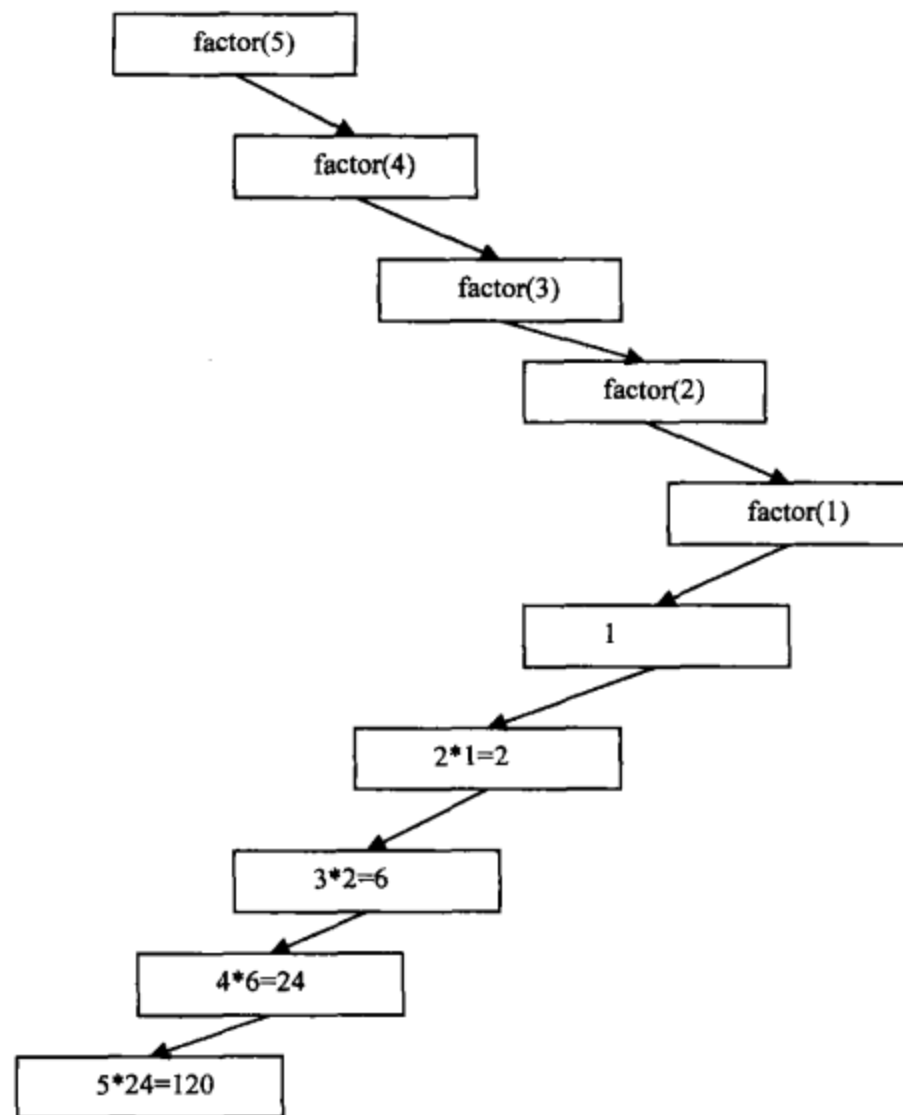


图 8-3 递归程序执行过程示意

8.14.3 递归程序设计方法

设计递归程序，一般可分为两个步骤。

- (1) 确定初始条件。这是递归能正常终止的依据，不允许出现无休止的递归。
- (2) 设计回推过程。这由问题本身的性质所决定。

例如，用递归的方法将输入的任意十进制整数按相反的顺序把各位数字打印出来。设计步骤如下。

(1) 分析：如果一个整数 n 只有一位，就可以将它直接打印出来。要确定这个整数只有一位，只要满足 $n/10=0$ 的条件，这就是初始条件。

(2) 对超过一位的整数，可以先取出它的最低位，方法是用 $n=n\%10$ 实现。在取出最低位的同时，用 $n=n/10$ 来降低该数的位数。

于是，就可以很容易地编写出递归程序 `prind()`。

```
void prind(long n)
{
    if (n<0)
        n=-n;

    putchar('-');
    putchar(n%10+'0');

    if ((n/=10)!=0)
        prind(n);
}
```

程序中，第一个 if 语句用来将负整数转换成正整数，并打印出负号；在输出数字字符时，用 `n%10+'0'` 来把一位数字转换成对应的 ASCII 码值，因为数字 0 的 ASCII 码值为 48，1 的 ASCII 码值为 49，即任一数字加上 48 后就是它的 ASCII 码值。

对初学者来说，设计递归程序应注意以下几点。

- 程序中一定要有 if 或类似的功能来中止递归调用，否则程序会无休止地递归下去。
- 在递归程序中，中止递归的条件语句 if 所影响的范围内不要再用循环语句，以免造成错误，因为递归本身就是一种循环。
- 比较简单的递归程序都可以用循环程序实现。

下面两个程序是实现一样的功能：

程序 1：

```
int fib(int n)
{
    int f;
    if (n==0||n==1)
        f=1;
    else
        f=fib(n-1)+fib(n-2);
    return f;
}
```

程序 2：

```
int fib(int n)
{
    int f,f0=1,f1=1,i;
    for (i=2;i<=n;i++)
    {
        f=f0+f1;
```



```
        f1=f0;
        f0=f;
    }
    return f;
}
```

这两个程序都是求菲波那契(Fibonacci)数列的,前者是递归程序,后者是递推程序。菲波那契(Fibonacci)数列的定义为:

```
1 (n=0)
fib(n)= 1 (n=1)
fib(n-1)+fib(n-2) (n>1)
```

由于递归程序在运行时需要反复做栈的操作因此效率低,能不用就尽量不用。

8.15 程序文件结构

8.15.1 头文件

许多有关C语言的书都不强调头文件的作用,并且编译器也并不强调函数声明,所以它在大部分时间内似乎是可要可不要的,除非要声明结构时。在C++中,头文件的使用变得非常清楚。它们对于每个程序开发是强制的,在它们中放入了非常特殊的信息:声明。头文件告诉编译器哪些库是可用的,因为对于.cpp文件能够不要源代码而使用库(只需要对象文件或库文件),所以头文件是存放接口规范的唯一地方。

头文件是库的开发者和它的用户之间的合同。它说:“这里描述的是库能做什么。”它不说如何做,因为“如何做”存放在.cpp文件中,开发者不需要分发这些描述“如何做”的源代码给用户。

该合同描述数据结构,并说明函数调用的参数和返回值。用户需要这些信息来开发应用程序,编译器需要它们来产生相应的代码。

编译器强迫执行这一合同,也就是要求所有的结构和函数在它们使用之前被声明,当它们是成员函数时,在它们被定义之前被声明。这样,就强制把声明放在头文件中,并把这个头文件包含在定义成员函数的文件和使用它们的文件中。因为描述库的单个头文件被包括在整个系统中,所以编译器能保证一致和避免错误。

为了恰当地组织代码和写有效的头文件,有一些问题必须知道。第一个问题是将什么放进头文件中。基本规则是“只声明”,也就是说,对于编译器只需要一些信息以产生代

码或创建变量分配内存。这是因为，在一个项目中，头文件也许会包含在几个处理单元里，而如果内存分配不止一个地方，则连接器会产生多重定义错误。头文件可包含及不宜包含的内容见表 8-1 和表 8-2。

表 8-1 头文件一般可包含的内容

头文件可包含的内容	示 例
类型声明	如 <code>enum COLOR{ / / ...}</code>
函数声明	如 <code>extern int fn(char s)</code>
内联函数定义	如 <code>inline char fn(char p){return *p++;}</code>
常量定义	如 <code>const float pi=3.14</code>
数据声明	如 <code>extern int m; extern int a[]</code>
枚举	如 <code>enum BOOLEAN{false, true}</code>
包含指令(可嵌套)	如 <code>#include</code>
宏定义	如 <code>#define Case break;case</code>
注释	如 <code>//check for end of file</code>

表 8-2 头文件不宜包含的内容

头文件不包含的内容	示 例
一般函数定义	如 <code>char fn (char p) { return *p++;}</code>
数据定义	如 <code>int a; int b[5]</code>
常量聚集定义	如 <code>const int c[]={1,2,3}</code>

这个规则不是非常严格的。如果在头文件中定义“静态文件”的一段数据(只在文件内可视)，在这个项目中将有这个数据的多个实例，编译器不会报错。基本上，不要在头文件中做在连接时会引起混淆的任何事情。

关于头文件的第二个问题是重复声明。C 和 C++都允许对函数重复声明，只要这些重复声明匹配，但决不允许对结构重复声明。在 C++中，这个规则特别重要，因为如果编译器允许对结构重复声明，而且这两个重复声明又不一样，那么应当使用哪一个呢？

重复声明问题在 C++中很少出现，因为每个数据类型(带有函数的结构)一般都有自己的头文件。但如果希望创建使用某个数据类型的另一个数据类型时，必须在另一个头文件中包含它的头文件。在整个项目中，很可能有几个文件包含同一个头文件。在编译期间，编译器会几次看到同一个头文件。除非做适当的处理，否则编译器将认为是结构重复声明。

典型的防止方法是使用预处理器隔离这个头文件。如果有一个头文件名为 FOO.H，一般用“名字分解”产生预处理名，以防止多次包含这个头文件。FOO.H 头文件的内部可以

如下定义。

```
#ifndef FOO_H_
#define FOO_H_
// Rest of header here ...
#endif // FOO_H_
```



注意：不用前导下划线，因为标准 C 用前导下划线指明保留标识符。

8.15.2 文件作用域

如果函数或变量的声明不是全局或局部的，而是具有文件作用域，这样声明的标识符的作用域开始于声明点，结束于文件尾。根据这个定义，静态全局变量和全局变量是文件作用域的，静态函数也是文件作用域的。

在头文件的作用域中所进行的声明，若该头文件被一个源文件嵌入，则声明的作用域扩展到该文件中，直到源文件结束。

8.15.3 多文件结构

源文件中含有包含头文件的预编译语句，经过预编译后产生翻译单元，该翻译单元以临时文件的形式存放在计算机中。之后编译，进行语法检查，产生目标文件(.obj)。若干个目标文件经过连接，产生可执行文件(.exe)。连接包括 C++库函数的连接和标准类库的连接。

许多小程序可以由单个源文件建立，它编译成一个目标文件，然后输给连接器，产生运行程序。这样的程序维护方便。如果修改了源文件中的任何函数，只需再次启动编译器即可。

大程序倾向于分成多个源文件，其理由如下。

- 避免一而再、再而三地重复编译函数。因为编译器总是以文件为单位工作的。如果一个文件中包含的函数太多，则由于被修改的函数总是少数的几个，所以大多数正确的函数都得重新编译一次。
- 使程序更加容易管理。可以将程序按逻辑功能划分，分解成各个源文件，便于程序员的任务安排以及程序调试。
- 把相关函数放到一特定源文件中。例如，所有输入函数放在一个源文件中。

用 C++建立项目时，通常要汇集大量不同的类型(带有相关函数的数据结构)。一般将每个类型或一组相关类型放在一个单独的头文件中，然后在一个处理单元中定义这个类型的函数。当使用这个类型时必须包含这个头文件，形成适当的声明。

本书中有时会使用这个模式，但如果例子很小，结构声明、函数定义和 `main()` 函数可以出现在同一个文件中，应当记住，这实际上使用的是隔离的文件和头文件。

8.15.4 外部存储类型

大型程序都是由多个源文件组成的。当一个文件中的程序段需要使用另一个文件中定义的变量或函数时，需要将数据或函数声明为外部的(`extern`)来进行沟通。即一个文件通过用 `extern` 关键字来声明一个与另一文件声明的同名变量(或函数)来获得对另一文件中变量(或函数)的使用权。用 `extern` 关键字修饰的变量(或函数)为外部存储类型的变量(或函数)，其格式为：

格式 1:

```
extern 数据类型 变量名;           //声明数据为外部的
```

格式 2:

```
extern 类型 函数名(参数)         //声明函数是外部的
```

说明:

- 带 `extern` 的变量说明是变量声明，不是变量定义。
- 所有函数声明一般都放在源文件的开始位置。
- 默认的函数声明或定义默认总是 `extern` 的。

例如：下面两个文件构成了一个程序，该程序由一个工程文件 `extern.prj` 定义，工程文件和源文件中的内容分别为：

```
//extern.prj
extern_1.cpp
extern_2.cpp
```

```
//extern_1.cpp
# include <iostream>
using namespace std;
```

```
void fn1();
void fn2();           //默认的函数声明或定义总是 extern 的
int n;
```

```
void main()
{
    n=3;
    fn1();             //fn1()函数的定义在本文件中
    cout<<n<<endl;
```



```

}
void fn1()
{
    fn2();           //fn2()函数的定义不在本文件中, 在下面的"extern_2.cpp"中
}

//extern_2.cpp
extern int n;         //n 由另一个源文件定义
void fn2()            //fn2()函数在另一个源文件中使用
{
    n=8;              //此处使用 n,n 已声明为外部的
}

```

文件 `extern_1.cpp` 中主函数 `main()`调用了函数 `fn1()`, 函数 `fn1()`调用了函数 `fn2()`。根据先定义后使用的原则, 在 `main()`函数的前面应有函数 `fn1()`的声明, 在函数 `fn1()`的定义之前应有函数 `fn2()`的声明。

由于默认的函数声明或定义总是 `extern` 的, 虽然文件 `extern_1.cpp` 中只是声明了函数 `fn2()`, 并未给出函数定义, 链接器在链接时还是会自动将 `extern_2.cpp` 中的函数定义与之匹配起来。

所以在文件 `extern_1.cpp` 中, 为了调用 `fn1()`和 `fn2()`函数, 在文件一开始声明的函数原型等价于:

```

extern void fn1();
extern void fn2();

```

在文件 `extern_1.cpp` 中定义了全局变量 `n` 以供 `main()`函数使用。函数 `fn2()`中要使用在 `extern_1.cpp` 中定义的全局变量 `n`, 因为默认的变量声明不是外部的, 故需在文件开头声明带 `extern` 的 `int n`, 它表示该变量不在本文件中分配空间, 而在程序的其他文件中分配空间(变量定义)。

假如一个程序由 10 个源文件构造而成, 每个源文件都必须访问一个全局变量。在这种情形下, 其中的 9 个文件必须把变量声明为 `extern`, 另外一个则不能。虽然在包含 `main()` 函数的源文件中分配变量是最合理的, 但实际上哪个文件真正分配该变量是无关紧要的。



注意: 带 `extern` 的变量说明是变量声明, 不是变量定义。如果共同使用的变量一次都没有定义, 或在各个文件中分别定义, 造成定义多次, 或声明的类型不一致, 都会造成直接或间接的错误。例如:

```

//file1.cpp
int a=5 ;
int b=6;

```



```
extern int c;
```

```
//file2.cpp
```

```
int a;                // error: 多次定义
extern double b;      //error: 类型不一致
extern int c;         //error: 无定义
```

在上面的代码中，两个源文件都以常规方式定义变量 `a`，没有一个显式声明 `extern`，这时，其行为将依赖于编译器。在 VC 中可以通过编译，但在连接时，会给出一个 “`int a already defined in file1.obj`(变量 `a` 的定义已经在前一个文件中定义过)” 的错误。但 BC 则将每个文件的变量定义都看作是全局静态变量，所以程序将会运行，但两个文件中的变量 `a` 是互不相干的。

两个文件中 `b` 的类型不一样，这时，VC 将在连接时报告一个 “`link error: unresolved external(未定的外部名)`” 错误，而 BC 却不能发现该错误而使程序错误地运行下去。

两个源文件又都声明变量 `c` 为外部的，这时，编译也不会发生问题，但连接时却会找不到该变量，产生一个连接错误，因为没有一个文件为该变量分配空间。

本章小结

本章详细地介绍了函数的使用方式，对用各种方式使用函数的方法进行了详细解释，深入地分析了函数的执行流程，对多文件组织结构也进行了细致的说明。由于使用多文件结构产生了变量更多的声明和使用方式，本章对这些问题也进行了详细阐述。

习 题

1. 设计一个加法函数，求两个整数的和。
2. 设计一个应用程序，至少由 3 个文件组成，其中一个是主函数所在的文件，而另外的文件提供函数的声明和实现。
3. 已知一个沙漠中，有 A、B 两个点，其中 A 点能产石油，B 点为储存点，AB 相距 100 公里。现假设用一辆车来运输油，已知该车每公里消耗油量为 N ，总载荷加上储备箱，该车一共可装 $100N$ 的油，因此不能直接装满油就开向 B 点，必须在中间设置临时储存点，这样才能把油运到 B。问如何设置临时点，设置多少个，才能保证运送油量与消耗量的比值最大。

第 9 章 指针和引用

本章内容:

- 指针的定义与使用。
- 指针作为函数的参数和返回值。
- 动态分配内存。
- 引用。

重点:

- 动态分配内存。
- 指针与函数的结合应用。

目的:

掌握内存的使用原则, 理解 C++ 的内存分配与回收机制, 利用指针和引用, 灵活地设计函数。

指针是一种用来操作地址的特殊数据类型, 它可以用于数组、作为函数参数, 更加可以用来内存访问和堆内存操作。指针是 C++ 中一个重要的内容, 并且功能强大, 但是指针操作又是很危险的。引用是 C++ 的另一个特性。学完本章要求读者能够很好地使用指针, 能够用指针给函数传递参数, 同时能够理解指针、数组和字符串之间的关系, 掌握指针和引用的用法, 理解引用和指针之间的区别。

9.1 指针的概念

任何变量都在内存中占有一块存储区域, 变量的值就存放在这块内存区域之中(寄存器变量不在内存中, 而是在 CPU 的寄存器中)。定义变量的过程实际上是为变量申请内存的过程。变量名其实是变量在内存中存储位置的一个符号描述, 在程序被编译为二进制代码时, 编译器会自动将变量名替换为相应的内存地址。对该变量内容的访问实际上是通过该地址存取其中内容的操作。变量内容与地址存储的关系如图 9-1 所示。

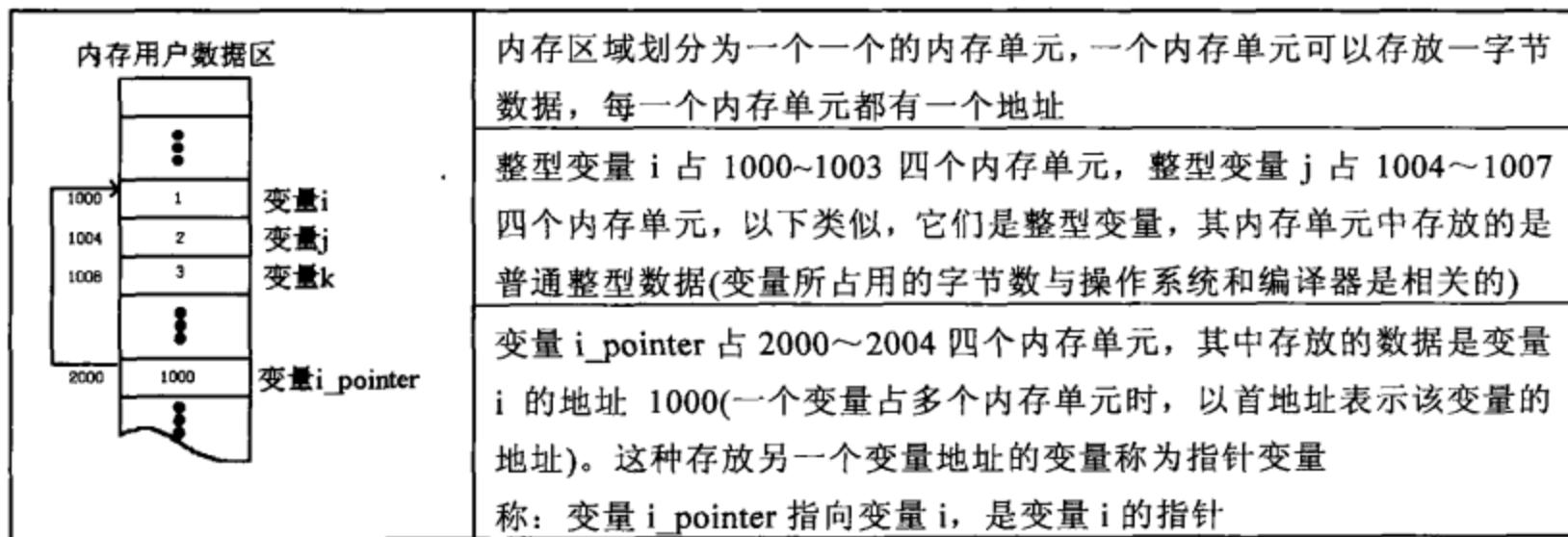


图 9-1 变量内容与地址存储的关系

打个比方，内存可以看作是一栋楼房，而内存单元是其中的房间，变量的地址相当与门牌号，房子里有人居住，就可以给这间房子取个名字，比如张三住在 101，那么就可以把 101 叫做“张三家”，当然也可以叫“心情小屋”等更好听的名字，这就类似于变量名，只要某人说“张三家”或“心情小屋”，就知道这指的是 101。

在程序设计中，变量名对应的内存单元的编号即为内存地址。

对于一个变量的访问(访问是指取出其值或向它赋值)方式有以下两种。

- 直接访问，通过变量名或内存地址直接访问，如通过变量名 i 直接访问。
- 间接访问，通过该变量的指针来访问，如通过 i_pointer 访问变量 i。

这两种访问方式的区别就好像要去张三家，如果知道张三家是 101 房间，那直接去就行了，这就是直接访问。当不知道门牌号时，就要想其他办法：先找到李四，李四知道张三家的地址，这样得到这个地址以后便可以去找张三了，这种通过其他人得到地址的方式就是上面说的间接访问。

在上面间接访问的例子中，可以看到需要李四这个中间人来提供地址，在程序中，通过一种特殊的变量即指针变量来实现李四的功能。

指针变量：是一个变量，其值是另一个变量的地址，一般是由数据类型后跟“*”，再跟指针变量名组成。当需要访问另一个变量时，可以通过指针变量中存放的内存地址来实现。通常，指针变量也称为指针。

指针变量(指针)和变量两者的关系如图 9-2 所示。

由图 9-2 可看出，指针变量中存放的是其他变量的内存地址，即其他变量的指针。如 pa 中存放的内容为 2600，该值对应于变量 a 的内存地址。

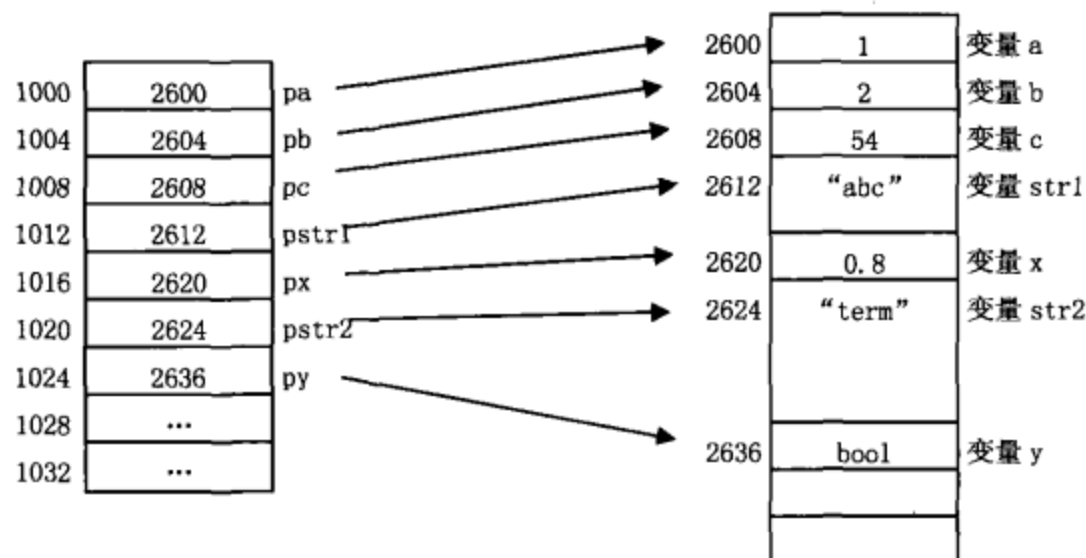


图 9-2 指针变量与变量在内存中的关系

9.2 指针声明和赋值

用下述方法来定义一个指针类型的变量。

```
int *ip;
```

此定义首先说明了它是一指针类型的变量，注意在定义中不要漏写符号“*”，否则它就为一般的整型变量了。另外，在定义中的 `int` 表示该指针变量为指向整型数的指针类型的变量，有时也可称 `ip` 为指向整数的指针。`ip` 是一个指针变量，它专门存放整型变量的地址。

指针变量的一般定义为：

类型标识符 *标识符；

其中标识符是指针变量的名字，标识符前加了“*”号，表示该变量是指针变量，而最前面的“类型标识符”表示该指针变量所指向的变量的类型(指针变量前的类型修饰符，决定了该指针能指向的变量类型，例如整型指针只能保存整型变量的地址)。一个指针变量只能指向同一种类型的变量，也就是说不能定义一个指针变量，既能指向一整型变量又能指向双精度变量。注意：指针变量前的类型标识符为指针所指变量或对象的类型，并不表示指针的类型。事实上，所有指针都是同一种类型：指针类型。

注意以下的定义：

```
int *a,b;
```

在这个定义中，变量 `a` 是一个指针，它可以指向整型变量，而变量 `b` 是个整型变量，

不是指针。a 前面的 “*” 仅仅修饰了 a 是个指针，没有修饰 b。也就是说，指针修饰符是与其后的变量结合，而不是与类型标识符结合。

指针变量同普通变量一样，使用之前不仅要定义说明，而且必须赋予具体的值。指针变量的赋值只能赋予地址，决不能赋予任何其他数据，否则将引起错误。在 C++ 语言中，变量的地址是由编译系统分配的，对用户完全透明，用户不能干涉变量的地址分配工作。C++ 语言中提供了地址运算符 & 来表示变量的地址。其一般形式为：

& 变量名；

如 &a 表示变量 a 的地址，&b 表示变量 b 的地址。变量本身必须预先说明。设有指向整型变量的指针变量 p，如要把整型变量 a 的地址赋予 p 可以有以下两种方式。

(1) 指针变量初始化的方法。

```
int a;  
int *p=&a;
```

注意，这里是用 &a 对 p 初始化，而不是对 *p 初始化。* 是用来标识这是一个指针变量的。

(2) 赋值语句的方法。

```
int a;  
int *p;  
p=&a;
```

不允许把一个数赋予指针变量，故下面的赋值是错误的：

```
int *p;  
p=1000;
```

前面说过，用户不能干涉变量地址的分配工作，因此地址 1000 用户不可以直接使用，必须经过系统进行地址分配，而这里直接将此地址给了指针 p，这是违反系统规定的。

当希望将一个变量的地址赋值给一个指针变量时，被赋值的指针变量前不能再加 “*” 说明符，如写为：

```
*p=&a;
```

这个表达式的结果将不是预期值。因为设计者的本意是将变量 a 的地址赋值给指针 p，但这样写的结果是将变量 a 的地址赋值给指针 p 所指向的空间。如果指针 p 在本次赋值之前已经初始化，例如指针 p 保存了另外一个变量 x 的地址，则这么做的结果是变量 x 的值被修改为变量 a 的地址；而如果指针 p 未经过初始化，则该语句在执行期间将抛出“指定地址不可写”错误。

当一个指针变量 **a** 保存了变量 **b** 的地址时，称指针 **a** 指向了变量 **b**。

和一般变量一样，对于外部或静态指针变量在定义中若不带初始化项，指针变量被初始化为 **NULL**，它的值为 0。C++ 中规定，当指针值为零时，指针不指向任何有效数据，有时也称指针为空指针。因此，当调用一个要返回指针的函数时，常使用返回值为 **NULL** 来指示函数调用中某些错误情况的发生。下面通过几个例子来说明对指针变量进行初始化的必要性。

其一：

```
#include <iostream>
using namespace std;
void main()
{
    int i,*p=&i;
    cout<<i<<endl;
    cout<<(*p)<<endl;
}
```

在这个例子中对指针变量作了初始化，输出结果将比较确定。

其二：

```
#include <iostream>
using namespace std;
void main()
{
    int i,*p;
    cout<<i<<endl;
    cout<<(*p)<<endl;
}
```

在这个例子中，由于 **p** 没有经过初始化，对于局部变量，编译器不会对其自动进行初始化，新定义的变量内容为分配的内存空间中原有的值，因此 **p** 指向的内存地址为一随机值。输出结果将无法确定。

在指针未经初始化时就对指针进行操作是严格禁止的，未经赋值的指针变量不能使用，否则将造成混乱，甚至系统崩溃。

其三：

```
#include <iostream>
using namespace std;
void main()
{
    int i,*p;
    cout<<i<<endl;
```

```

    *p = 15;
}

```

注意，在这个例子中，没有对指针初始化，因而 *p* 的值可能任意。根据操作系统的内存分配原理可知，计算机在内存的低地址空间存放的是操作系统运行需要的数据，当未经初始化的指针错误地指向了这批数据区，而程序又对这段数据进行了写操作以后，造成的后果将是难以预测的。

9.3 通过指针访问数据

定义并为指针赋值以后，可以通过指针对其指向的数据进行存取操作，操作格式为：

*指针变量名

“*”是取内容运算符，单目运算符，其结合性为自右至左，用来表示指针变量所指的变量。在“*”运算符之后跟的变量必须是指针变量。需要注意的是指针运算符“*”和指针变量说明中的指针说明符“*”不是一回事。在指针变量说明中，“*”是类型说明符，表示其后的变量是指针类型。

对指针指向的数据操作可以分为读和写两种，下面分别加以说明。

1. 读取数据

例如：

```

#include <iostream>
using namespace std;
void main()
{
    int a=105,*p=&a;
    cout<<"变量 a 的值是: "<<a<<endl;
    cout<<"指针 p 所指向的内存的数据是: "<<(*p)<<endl;
}

```

在本例中定义了一个变量并将其值赋为 105，然后定义了指针变量 *p* 并将其值赋为前面定义的变量的地址，即将指针指向了该变量。程序运行结果为：

```

变量 a 的值是: 105
指针 p 所指向的内存的数据是: 105

```

2. 写数据

例如：

```

#include <iostream>

```

```

using namespace std;
void main()
{
    int a=100, *p=&a;
    cout<<" 变量 a 的值是: " <<a<<endl;
    *p = 80;
    cout<<"指针 p 所指向的内存的数据是: " <<(*p)<<endl;
    cout<<"现在变量 a 的值是: " <<a<<endl;
}

```

在本例中输出变量 a 的值后通过指针变量修改了 a 的值。程序运行结果为：

变量 a 的值是：100

指针 p 所指向的内存的数据是：80

现在变量 a 的值是：80



注意： 前面讲到的指针变量定义中的 *p 与对数据操作时使用的 *p 是两个完全不同的概念。前者只是标识定义的是一个指针变量，而后者则是说明要对指针指向的数据进行操作，而不是对指针变量本身进行操作。

图 9-3~图 9-5 是对指针和指针指向的地址赋值过程的示意图。

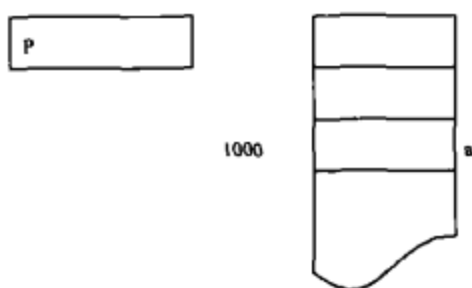


图 9-3 原有数据

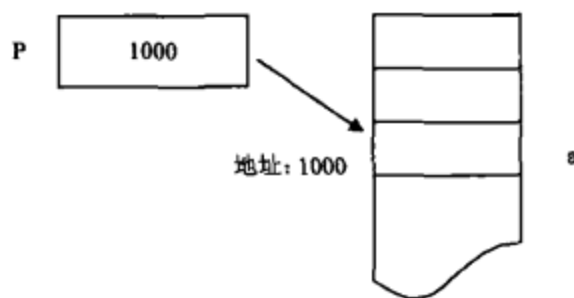


图 9-4 执行完 p=&a 以后

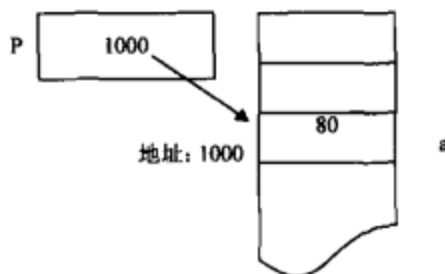


图 9-5 执行完(*p)=80 以后

由这三个图可以很明显地看出，p=&a 执行的是将 a 的地址赋给指针变量 p 的操作，此时便可以说指针 p 指向变量 a；(*p)=80 是对 p 指向的地址中内容的操作，亦即对 a 进行的赋值操作。

下面是一个用指针访问变量进行运算的例子。


```

#include <iostream>
using namespace std;
main(){
    int a=10,b=20,s,t,*pa,*pb; //说明 pa,pb 为整型指针变量
    pa=&a;                      //给指针变量 pa 赋值, pa 指向变量 a
    pb=&b;                      //给指针变量 pb 赋值, pb 指向变量 b
    s=*pa+*pb;                 //本行的意义是求 a+b 之和 (*pa 就是 a, *pb 就是 b)
    t=*pa**pb;                 //本行是求 a*b 之积
    cout<<"a="<<a<<"b="<<b<<"a+b="<<a+b<<"a*b="<<a*b<<endl;
    cout<<"s="<<s<<"t="<<t<<endl; //输出结果
}

```

9.4 指针运算

指针运算包括指针变量的加减运算和两个指针变量之间运算。

1. 指针变量的加减运算

对于指向数组的指针变量,可以加上或减去一个整数 n 。设 pa 是指向数组 a 的指针变量,则 $pa+n$ 、 $pa-n$ 、 $pa++$ 、 $++pa$ 、 $pa--$ 、 $--pa$ 运算都是合法的。指针变量加或减一个整数 n 的意义是把指针指向的当前位置(指向某数组元素)向前或向后移动 n 个位置。应该注意,数组指针变量向前或向后移动一个位置和地址加 1 或减 1 在概念上是不同的。因为数组可以有不同的类型,各种类型的数组元素所占的字节长度是不同的。如数组指针变量加 1,即向后移动 1 个位置表示指针变量指向下一个数据元素的首地址,而不是在原地址基础上加 1。例如:

```

int a[5],*pa;
pa=&a[0]; //pa 指向 a[0]
pa=pa+2; //pa 指向 a[2], 即 pa 的值为&a[2]

```

假设原 pa 的值是 1000,这个加法操作后 pa 的值不是 1002,而是 1008(在 32 位编译器下)。这里的加 2 是将指针 pa 的值加上两个整型变量所占的字节数。

图 9-6 为整型指针加 1 后的结果:指针移动了 4 个字节(在 32 位机器中)。

下面再看一个双精度指针的例子。

```

double db,*dp;
dp = &db;
dp = dp + 1;

```

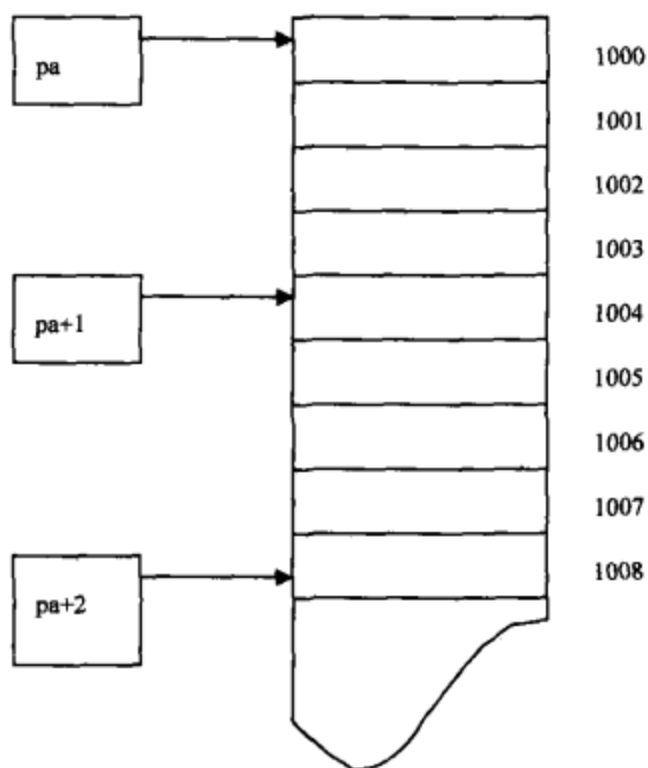
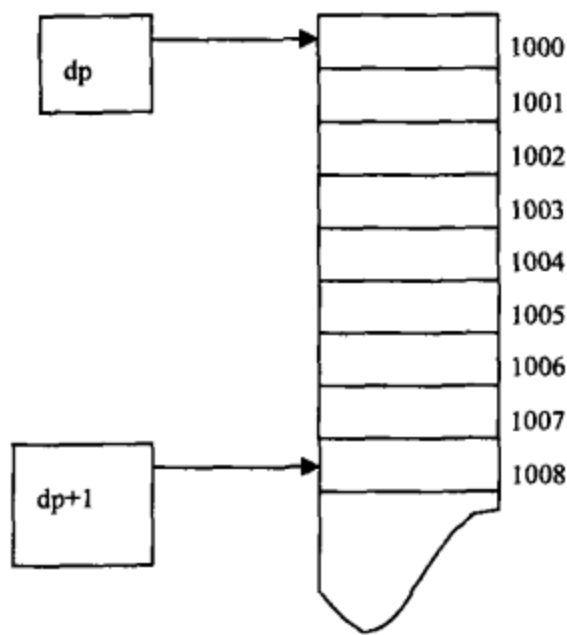


图 9-6 整型指针加 1 的结果

由于在 32 位机器中, `double` 类型的数据占据 64 位数据空间, 所以 `double` 类型的指针在加 1 时需向后移动 8 个存储单元, 如图 9-7 所示。

图 9-7 `double` 类型指针加 1 后的结果

注意: 指针变量的加减运算只能对数组指针变量进行, 对指向其他类型变量的指针变量作加减运算结果将不可预期(若把其他类型的做数组处理则另当别论, 在实际工作中有为了对大块内存数据进行分割处理而作这样的应用)。

2. 两个指针变量之间的运算

只有拥有相同类型修饰符的两个指针变量之间才能进行运算，否则运算毫无意义。

1) 两指针变量相减

两指针变量相减所得之差是两个指针所指数组元素之间相差的元素个数，实际上就是两个指针值(地址)相减之差再除以该数组元素的长度(字节数)所得的结果。例如，pf1 和 pf2 是指向同一浮点数组的两个指针变量，设 pf1 的值为 2016，pf2 的值为 2000，而浮点数组每个元素占 4 个字节，所以 pf1-pf2 的结果为(2016-2000)/4=4，表示 pf1 和 pf2 之间相差 4 个元素。两个指针变量不能进行加法运算。例如，pf1+pf2 毫无实际意义。

2) 两指针变量进行关系运算

指向同一数组的两指针变量进行关系运算可表示它们所指数组元素之间的关系。例如：

- pf1==pf2 表示 pf1 和 pf2 指向同一数组元素。
- pf1>pf2 表示 pf1 处于高地址位置。
- pf1<pf2 表示 pf2 处于低地址位置。

指针变量还可以与 0 比较。设 p 为指针变量，则 p==0 表明 p 是空指针，它不指向任何变量；p!=0 表示 p 不是空指针。空指针是由对指针变量赋予 0 值而得到的。例如：

```
#define NULL 0
int *p=NULL;
```

对指针变量赋 0 值和不赋值是不同的。指针变量未赋值时可以是任意值，是不能使用的，否则将造成意外错误。而指针变量赋 0 值后，则可以使用，只是它不指向具体的变量而已。

下列是指针应用的几种情形。

```
#include <iostream>
using namespace std;
main()
{
    int a,b,c,*pmax,*pmin;           //pmax,pmin 为整型指针变量
    cout<<"input three numbers: "<<endl; //输入提示
    cin>>a>>b>>c;                   //输入三个数字
    if(a>b)
    {
        pmax=&a;                     //如果第一个数字大于第二个数字
        pmin=&b;                     //指针变量赋值
    }
    else
```

```

    {
        pmax=&b;                                //指针变量赋值
        pmin=&a;                                //指针变量赋值
        if(c>*pmax)
            pmax=&c;                            //判断并赋值
        if(c<*pmin)
            pmin=&c;                            //判断并赋值
        cout<<"max="<<(*pmax)<<"min="<<(*pmin)<<endl; //输出结果
    }
}

```

9.5 动态内存分配

堆(heap)是内存空间。堆是区别于栈区、全局数据区和代码区的另一个内存区域。堆允许程序在运行时(而不是在编译时), 申请某个大小的内存空间。

在通常情况, 一旦定义了一个数组, 那么不管这个数组是局部的(在栈中分配)还是全局的(在全局数据区分配), 它的大小在程序编译时即是已知的, 因为必须用一个常数或常量对数组的大小进行声明, 如下面的程序所示。

```

int i=10;
//...
int a[i]; //error: 定义时不允许数组元素的个数为变量
int b[20]; //ok

```

但是, 在编写程序时不是总能知道数组应该定义成多大, 例如第8章8.13节中的栈, 如果数组定义少了, 可能造成程序无法计算, 而如果定义数组元素多了就浪费内存了, 更何况有时候根本不知道需要使用多少个数组元素。因此, 需要在程序运行时从系统中获取内存; 当无须指针变量操作时, 可以将其所分配的内存归还系统, 此过程称之为内存单元的释放。

程序在编译和连接时不予确定这种在运行中获取的内存空间, 这种内存环境随着程序运行的进展而时大时小的内存就是堆内存, 所以堆内存是动态的。堆内存也称动态内存。

另外, 由前述可知, 指针变量指向的变量通过操纵地址的方式来改变数据。指针变量的取值是内存的地址, 这个地址应当是安全的, 不可以是随意的, 否则, 写入内存单元的值将会使得已存放的数据或程序丢失。应使用编译系统提供的标准函数来实现地址分配。

9.5.1 malloc()和 free()函数

ANSI 标准建议设置了两个最常用的动态分配内存的函数 `malloc()` 和 `free()`，函数在 `alloc.h` 文件中声明。`malloc()` 函数用以向编译系统申请分配内存；`free()` 函数用以在使用完毕释放掉所占内存。

1. malloc()函数

函数 `malloc()` 的原型如下：

```
void * malloc(size_t size);
```

其中 `size` 指定的是需要申请的内存的字节数。

用 `malloc` 函数申请一块 `length` 个整数类型的内存，程序如下：

```
int *p = (int *) malloc(sizeof(int) * length);
```

把注意力集中在两个要素上：“类型转换”和“`sizeof`^①”运算符。

- `malloc()` 函数返回值的类型是 `void *`，所以在调用 `malloc()` 函数时要显式地进行类型转换，将 `void *` 转换成所需要的指针类型。
- `malloc()` 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。用户通常不去记忆数据类型的变量在用户正在使用的平台下的确切字节数。例如 `int` 变量在 16 位系统下是 2 字节，在 32 位下是 4 字节；而 `float` 变量在 16 位系统下是 4 字节，在 32 位下也是 4 字节。最好用以下程序作一次测试：

```
cout << sizeof(char) << endl;
cout << sizeof(int) << endl;
cout << sizeof(unsigned int) << endl;
cout << sizeof(long) << endl;
cout << sizeof(unsigned long) << endl;
cout << sizeof(float) << endl;
cout << sizeof(double) << endl;
cout << sizeof(void *) << endl;
```

以上语句中 `sizeof` 运算符用于返回某个数据类型所占的字节数。

2. free()函数

函数 `free()` 的原型如下：

① 在 `malloc` 的“()”中使用 `sizeof` 运算符是良好的风格。

```
void free( void * memblock );
```

为什么 free() 函数不像 malloc() 函数那样复杂呢？这是因为指针 p 的类型以及它所指的内存的容量事先都是知道的，语句 free(p) 能正确地释放内存。如果 p 是 NULL 指针，那么 free() 函数对 p 无论操作多少次都不会出问题；如果 p 不是 NULL 指针，那么 free() 函数对 p 连续操作两次就会导致程序运行错误。

以下示例实现两个字符串的交换。

```
#include <string >
#include <iostream>
#include <malloc>
using namespace std;
main()
{
    char *ptr1,*ptr2,*temp;
    ptr1=malloc(30);           //动态为指针变量分配长度为 30 字节的存储空间
    ptr2 = malloc(20);
    temp = malloc(30);
    cout<<"请输入字符串 1:"<<endl;
    gets(ptr1);               //输入字符串
    cout<<"请输入字符串 2:";
    gets(ptr2);
    cout<<"字符串 1 - - - - - 字符串 2"<<endl;
    cout<<ptr1<<" . . . . . "<<ptr2<<endl;
    strcpy(temp,ptr1);        //串复制
    strcpy(ptr1,ptr2);
    strcpy(ptr2,temp);
    cout<<"字符串 1 - - - - - 字符串 2"<<endl;
    cout<< ptr1<<" . . . . . "<<ptr2<<endl;
    free(ptr1);
    free(ptr2);
}
```

为指针变量分配的存储空间长度取决于存放字符的多少。在上述的程序中，两个串的交换可以通过标准函数 strcpy() 来完成，也可以通过串指针交换指向完成，用 temp = ptr1; ptr1 = ptr2; ptr2 = temp; 三条赋值语句实现。但是，利用指针交换指向，其物理意义与串通过函数进行的复制完全不同。前者是存放串地址的指针变量数据交换，后者是串在内存物理空间的数据交换。指针变量用完后，将指针变量所占的存储空间释放。

程序运行情况如下：

请输入字符串 1：

Test1

请输入字符串 2：

```

Test2
str1 - - - - - str2
Test1- - - - - Test2
str1 - - - - - str2
Test2 - - - - - Test1

```

上例中并没有保证一定可以从堆中获得所需内存。有时，系统能提供的堆空间不够分配，这时系统会返回一个空指针值 `NULL`。此时所有对该指针的访问都是破坏性的，因此调用 `malloc()` 函数更完善的代码应该如下：

```

if((p=(int *)malloc(arraysize*sizeof(int))==NULL)
{
    cout<<"内存分配失败\n";
    exit(1);
}

```

9.5.2 new 和 delete 运算符

`new` 和 `delete` 是 C++ 新增的运算符，它们提供了存储的动态分配和释放功能，其作用相当于 C 语言的 `malloc()` 和 `free()` 函数，但是性能更为优越。使用 `new` 运算符较之使用 `malloc()` 函数有以下几个优点。

- `new` 运算符自动计算要分配类型的大小，不需使用 `sizeof` 运算符，比较省事，可以避免错误。
- `new` 运算符自动地返回正确的指针类型，不用进行强制指针类型转换。
- 可以用 `new` 运算符对分配的对象进行初始化。
- `malloc()` 函数在分配空间时，不会调用对应类型的构造函数^①，而 `new` 运算符调用相应的构造函数。同理 `free()` 函数不会调用析构函数而 `delete` 运算符会调用。

用 `new` 运算符申请一块长度为整数类型的内存，程序如下。

```

int *p1 = new int;           // 申请一个整型变量大小的内存空间
int *p = new int[length];   // 申请一块 length 个整数类型的内存

```

运算符 `new` 使用起来要比函数 `malloc()` 简单得多，这是因为 `new` 内置了 `sizeof`、类型转换和类型安全检查功能。

使用 `delete` 释放内存的格式如下。

① 构造函数：在第 11 章“类与对象”中进行详细的讲述，这一条说明是对类或结构体来说的。

(1) 释放单个变量的内存。

释放单个变量内存的格式为：

delete 指针变量

例如：

```
int *p1 = new int;           // 申请一个整型变量大小的内存空间
delete p1;                   // 释放 p1 指向的内存
```

(2) 释放作为数组分配的内存。

释放作为数组分配的内存，需要在指针变量后加[]，其中不必注明内存大小，其格式为：

delete 指针变量[];

例如：

```
int *p2 = new int[length]; // 申请一块 length 个整数类型的内存
delete p2[]                  // 释放 p2 数组指向的内存
```

由此例可看出 new 的返回值无须显式转换类型，直接赋给整数指针 array；new 的操作数是 int[arraysize]，它只要指明什么类型和要几个元素就可以了，它比 malloc()函数更简洁。

9.5.3 指针与数组

如果指针只分配了一个对应数据类型长度的空间，则该指针指向的空间就相当于一个变量，例如：

```
int *p = new int;
```

此时，*p 就相当于一个整型变量。

如果指针分配了多个对应数据类型的空间，又是什么情况呢？例如：

```
int *p = new int[10];
```

这时候，指针 p 指向了新开辟的 10 个整型空间中第一个空间的首地址，如果想使用第一个空间存储数据，则可以写成：

```
*p = 3;
```

如果想使用第 2 个空间存储数据，则可以写成：

```
*(p+1) = 4;
```

这里涉及指针运算，请参考 9.4 节。1 代表向后移动一个整型空间那么多，以此类推可

知最后一个空间的访问方式是 $*(p+9)$ 。进一步思考， $*p$ 就是 $*(p+0)$ 。

这样使用，从语法角度说完全正确，但是从开发角度来说，可读性就不好了，因此还可以采用第二种写法来访问变量，即：

```
*(p+0) 可以写成 p[0];
*(p+1) 可以写成 p[1];
*(p+2) 可以写成 p[2];
*(p+n) 可以写成 p[n];
```

看起来好熟悉，这不就是数组么，没错！这正是数组元素的访问方式。换句话说，如果一个指针，一次性的分配了多个对应类型的空间，则该指针指向的空间可以看作一个数组。其访问方式可以完全按照数组的方式进行访问。

$p[i]$ 表示数组的第 i 个元素的值，而 $p+i$ 表示第 i 个元素的地址，对其间接访问，则 $*(p+i)$ 就表示第 i 个元素的值。另外，下标操作是针对地址而不仅仅是针对数组名的。上面的式子等价的事实使得数组与指针相互转换非常灵活。相关的程序示例如下：

```
#include <iostream.h>
//无须头文件#include <alloc.h>
void main()
{
    int arraysize; //元素个数
    int *array;
    cout <<"请输入要开辟的数组的大小:\n";
    cin >>arraysize;
    if((array=new int[arraysize])==NULL){ //分配堆内存
        cout<<"内存分配失败.\n";
        return ;
    }
    for(int count=0; count<arraysize; count++)
        array[count]=count*2;
    for(int count=0; count<arraysize; count++)
        cout <<array[count] <<" ";
    cout <<endl;
    delete[]array; //释放堆内存
}
```

程序运行情况如下：

请输入要开辟的数组的大小：

10

0 2 4 6 8 10 12 14 16 18

基于上述使用过程可知：数组名可以拿来初始化指针。

例如：

```
int a[10];
int *p = a;
```

相应的&a[i]等价于 a+i 等价于 p+i 等价于&p[i]。

其实，一维数组的数组名本身就是指针，它的类型是指向数组元素的指针。&a[i]表示数组第 i 个元素的地址。

例如，对于数组的求和运算，可以有下面 5 种方法。

```
#include <iostream>
using namespace std;
int sum1,sum2,sum3,sum4,sum5;           //存放每种方法的结果
int iArray[]={1,4,2,7,13,32,21,48,16,30}; //全局数组
int* iPtr;
void main()
{
    int size,n;
    size=sizeof(iArray)/sizeof(*iArray); //元素个数
    for(n=0; n<size; n++) //方法 1
        sum1 += iArray[n];
    iPtr=iArray;
    for(n=0; n<size; n++) //方法 2
        sum2 += *iPtr++;
    iPtr=iArray; //此句不能省略，因为方法 2 修改了 iPtr
    for(n=0; n<size; n++) //方法 3
        sum3 += *(iPtr+n);
    iPtr=iArray; //此句可以省略，因为方法 3 没有修改 iPtr
    for(n=0; n<size; n++) //方法 4
        sum4 += iPtr[n];
    for(n=0; n<size; n++) //方法 5
        sum5 += *(iArray+n);
    cout <<sum1 <<endl
        <<sum2 <<endl
        <<sum3 <<endl
        <<sum4 <<endl
        <<sum5 <<endl;
}
```

程序运行结果如下：

```
174
174
174
174
```

程序中求元素个数的表达式中, `sizeof(*iArray)`表示数组元素类型所占的字节数, 它可以表示成 `sizeof(int)`。没有这样做的原因是, 该语句可以适应 `float` 或 `double` 等类型的数组, 任凭全局数据的类型如何变化, 主函数中的语句都不用作修改。

一般来说, 在机器指令的实现上, 指针表示不比下标表示效率低。所以, `*(iPtr+n)`(指针表示)或 `*(iArray+n)`的表示总是不差于 `iPtr[n]`(下标表示)或 `iArray[n]`的表示, 但从可读性来说, 后者似乎优于前者。

一维数组的名字, 就是指向数组第一个元素地址的指针, 也称指向数组的首地址, 那么多维数组的名字呢?

这里以二维数组为例进行讨论。

假设有二维数组 `a[10][5]`, 数组本身可以理解为一个一维的数组 `x[10]`。`x[10]`与 `int y[10]`不同的是, `y` 中的每个元素是 `int` 类型的, 而 `x` 中的每个元素, 都是一个有 5 个元素的整型数组。对于数组中的元素 `a[1][4]`, 可以理解为数组 `x` 中第 2 个元素内的第 5 个小元素, 所以 `a[1][4]`可以写成如下形式:

`a[1][4]`等价于 `*(*(a+1)+4)`

对于任意的下标 `m`、`n`, `a[m][n]`等价于 `*(*(a+m)+n)`, 当 `m`、`n` 的值都是 0 的时候, 就变成 `a[0][0]=*(*(a+0)+0)=**a`。注意, `a` 的前面有两个*, 这说明, `a` 是指向 `a[0][0]`的指针的指针, 是二重指针^①。

所以对于多维数组, 数组的名字是多重指针。

数组名是指针常量, 区别于指针变量, 所以, 给数组名赋值是错误的。

例如, 下面的代码对数组求和, 企图以数组名的增量来实现。

```
int a[10];
int sum=0;
//...
for(int i=0;i<10;i++)
{
    sum+=a;
    a++; //error: 数组名不是左值
}
```

由于指针常量不是左值, “`a++;`”意味着“`a=a+1;`”, 即要求 `a` 是一个左值。所以, 在

① 参见 9.12 节指向指针的指针。

上例中，BC 编译器将给出“Lvalue required”的错误，VC 编译器将给出“leftoperand must be an lvalue”的错误。

对于编译器来说，数组名表示内存中分配了数组的固定位置，修改了这个数组名，就会丢失数组空间，所以数组名所代表的地址不能被修改。

9.6 动态内存分配的应用

9.6.1 应用举例 1

思考第 8 章第 13 节中的栈，这个栈的一个明显缺陷就是最多仅能存储 200 个数据，对不同的应用来说，需要存储的数据可能不尽相同，有的程序可能只需要 20 个空间，而有的程序可能会需要 1000 个空间，按照 8.13 节中的定义方式，很明显适用性不高，而应用动态分配内存的方式，就可以大大提高灵活性。因此程序修改为如下方式工作。

- 将全局数组 data 变为一个指针，并增加一个全局变量 size 备用。
- 增加一个函数 init(int length)，用于指定需要栈空间的大小。
- 增加一个函数 del()，用于释放动态分配的内存。

最后实现的代码如下：

```
int *data = 0;
int pos = 0;
int size;
bool init(int length)
{
    size = length;
    data = new int[length];
    if(data == NULL)
    {
        return false;
    }
    else
        return true;
}
void del()
{
    if(data != 0)
    {
        delete []data;
    }
}
```



```
}
bool push(int a)
{
    if(pos==size)
    {
        return false;
    }
    data[pos++] = a;
    return true;
}
int pop()
{
    return data[--pos];
}
```

这样修改后，调用部分可修改成如下模式：

```
...
int len;
cin>>len;
if(!init(len))
{
    cout<<"栈内存分配失败！"<<endl;
}
...
```

程序已经有些灵活了，但还并不完善，在后续的章节中会不断完善它。

9.6.2 应用举例 2

在游戏程序中，一切显示效果都离不开图像，因此程序一定要能够读取图像文件的数据并能够进行显示，本节将描述如何显示一幅 BMP 图像，期间可能涉及文件的操作，读者可先参考一下相关书籍中有关文件操作的知识，这里的应用比较简单，不进行详细的描述。

为了能够编写图形显示程序，首先来研究以下相关的基本知识。

1. 计算机中颜色的表示

在计算机中表示颜色的方式有很多种，但在游戏程序设计中，使用最多的是采用 RGB 方式表述颜色，即一个颜色被分解为红(R)、绿(G)、蓝(B)三种颜色。任何一种颜色，都是由这三种颜色按每种分量的不同饱和度^①混合成的。例如黄色，可以由红和绿两种颜色按照

① 颜色的饱和度：在计算机中，表示颜色的程度，用从 0~255 的数据来表示，0 表示不包含颜色，而 255 表示该颜色的最大浓度。在实际工作中，也可以把 0~255 映射到 0~1.0 范围内的浮点型数。

相同的饱和度混合而成。如图 9-8 所示。

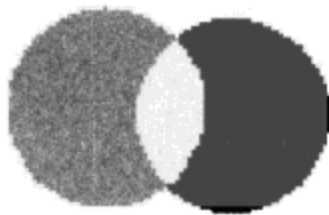


图 9-8 混合黄色

2. 图像在计算机中的存储方法

所谓图像，就是连续的颜色点组成的一个方阵，只要点足够小，用肉眼分不出两个点之间的距离，并且颜色过渡平滑，就可产生质量很高的图像。例如图 9-9，左边的小图是原始的图片，但如果放大该图，就产生图中右边的显示效果。从右图中可以看到，其实图就是由一系列的点构成，每个点都有颜色，相邻点的颜色大部分情况是比较接近的，因此产生的图像过渡平滑，效果很好。在计算机中只要将每个点的位置和颜色记录下来，就等价于记录了该图像。



图 9-9 原始图和放大图

3. BMP 文件的结构

严格说来，BMP 文件有很多小的细节区别，这是一本关于 C++的书，因此对 BMP 的结构这里不准备深入分析，而仅以标准的、无压缩的 24 位 BMP 图像为例，简单说明一下这样的文件结构，目的是为了最终能读出其中的数据，最后显示出来。

对于无压缩的 24 位 BMP 图像来说，其大致结构如下：

文件开始的第 1 和第 2 字节，永远是 BM 两个数据；第 10、11 字节构成一个短整型，表示的是图形数据的开始位置(例如，如果该数据为 54，则表示该文件从第 54 字节开始才是真正的图形数据)；第 18、19 字节表示的是该图的宽度(以像素为单位描述)；第 22、23 字节表示的是该图的高度，从指定的位置(例如 54)开始，就是图像的数据。上文说了，图像的数据就是每一点的颜色，而每点的颜色由 3 个无符号的字节数据构成，分别表示红、绿、蓝分量。首先获得图像的宽度和高度，从而可知道有多少个图像点，之后就可以读取

数据并显示。

在掌握上述 3 个基本知识点之后，就可以开始编写程序了，但是有一个问题需要考虑一下：所有点的颜色数据都需要在内存中保存，因此需要定义一个数组来保存数据，可是，数组定义多大才合适呢？在编写程序的时候无法确定，因为不同的图大小是不同的。所以这里使用指针来解决问题。程序设计思路如下：

```
void readbmp()
{
    unsigned char *data;
    short pos;
    short width,height;
    unsigned char r,g,b;
    int i,j;
    int arraypos = 0;    //这个数据用来表示数据应该存入数组的第几个字节
    FILE *fp = fopen("test.bmp","rb");    //打开 BMP 文件
    fseek(fp,10,SEEK_SET);    //定位文件读写位置到第 10 字节
    fread(&pos,sizeof(short),1,fp);    //读取一个短整型数据，存至 pos 中备用
    fseek(fp,18,SEEK_SET);    //定位文件读写位置到第 18 字节
    fread(&width,sizeof(short),1,fp);    //读取一个短整型数据，存至 width 中
    fseek(fp,10,SEEK_SET);    //定位文件读写位置到第 22 字节
    fread(&height,sizeof(short),1,fp);    //读取一个短整型数据，存至 height 中
    data = new unsigned char[width*height*3];
        //至此，已经知道图像的宽度和高度，可以分配内存了
    fseek(fp, pos,SEEK_SET);    //定位文件读写位置到第 pos 字节

    for (i= 0; i< width; i++)
    {
        for(j = 0; j<height; j++)
        {
            fread(&b,sizeof(unsigned char) ,1,fp);
            //连续读 3 个字节的数据，分别存入 b,g,r 变量中，注意 BMP 文件是按照
            //b,g,r 的次序存储的颜色数据
            fread(&g, sizeof(unsigned char) ,1,fp);
            fread(&r, sizeof(unsigned char) ,1,fp);
            data[arraypos++] = r;    //每存储一个数据，arraypos 加一个 1
            data[arraypos++] = g;
            data[arraypos++] = b;
        }
    }
    //这之后，可以使用 data 数组中的数据进行显示等操作
    delete []data    //使用完成后，释放 data 的空间
}
```

当数据读入内存后，唯一需要解决的就是显示它，但是标准 C++ 没有定义图形函数，

所有的应用程序都是工作在字符模式下的，因此只能利用其他的图形 API 来进行绘制。在第 2 章结尾的时候曾经介绍过 OpenGL 图形 API 系统，那么现在就用 OpenGL 来绘制这个 BMP 文件。

首先来看利用 OpenGL 绘图的基本流程，分为以下 4 步。

- ① 初始化一个 OpenGL 应用程序窗口。
- ② 指定由哪个函数进行窗口大小等信息的处理。
- ③ 指定哪个函数可以向窗口上绘制图形。
- ④ 绘制图形。

1) 初始化一个 OpenGL 应用程序窗口

创建窗口的方式有很多，但直接利用 C++ 是无法创建窗口的，另外，窗口大多数与具体的操作系统有关系，所以这里采用 OpenGL 的辅助工具 GLUT 创建窗口。创建窗口时需要指定窗口的属性。

定义如下变量并给定值：

```
int DispModel = GLUT_RGBA|GLUT_DOUBLE①;
//初始化 GLUT 环境
glutInit(&argc,argv);
//指定窗口是黑色的背景
glClearColor(0.0,0.0,0.0,1.0);
//将指定的窗口模式通知给 OpenGL
glutInitDisplayMode(DispModel);
//创建窗口
glutCreateWindow("show bmp");
```

至此，一个 OpenGL 窗口就创建好了。但是现在还不能显示图形，还要再做一些工作才可以绘制图形。

2) 指定由哪个函数进行窗口大小等信息的处理

首先定义一个函数，函数如下：

```
void myshape(int w,int h)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
```

① GLUT_RGBA 等数据都是枚举类型，对于这些值的含义，读者在这里不用仔细研究，只要知道这是用来指定窗口属性的就可以了。在本书中，这些值一直不变，有兴趣的读者可参考 OpenGL 手册相关章节。


```

    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) width, 0.0, (GLdouble) height);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

```

在实际工作中函数的名字是随意的,但是参数类型和返回值类型不可以变更。函数体的内容读者一样不需要去研究过程,仅需要知道当窗口变化的时候该函数需要被调用就可以了,代码写在这里,是为了保证读者写出的程序一定可以运行。在本书中,该函数的内容是保持不变的。编写完该函数后,可以接着 `glutCreateWindow` 这条语句继续写程序了:

```

glutReshapeFunc(myshape); //指定窗口变化时需要的函数,请注意该语句的参数就是
                           刚刚创建的那个函数

```

3) 指定哪个函数可以向窗口上绘制图形

同第2个步骤一样,首先创建如下一个函数。

```

void display()
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    //绘图的代码,就写在这里
    glFlush();
    glutSwapBuffers();
}

```

然后继续编写主程序:

```

glutDisplayFunc(display); //指定由 display 函数负责绘制图形
glutMainLoop(); //让 OpenGL 开始工作

```

现在,OpenGL 可以开始工作了。图形的数据此时还在内存数组中,并没有绘制到屏幕上去,要想绘制上去,还需要学习以下两个函数。

```

glRasterPos2i(m_dPosX,m_dPosY); //指定绘图的位置

```

其中 `dPosX` 和 `dPosY` 是指所绘制图形的屏幕坐标,该坐标是以 GLUT 窗口的左下角为坐标原点的一个标准笛卡儿坐标系中的坐标值。

```

glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE,databuf);

```

其中 `width` 是指将要绘制的图形的宽度,而 `height` 是指将要绘制的图形的高度, `databuf` 是保存了将要绘制的图形的数据的数组。

只要在 `display` 函数内调用上述两个函数,就可以绘制图形了,本例代码如下:

```

glRasterPos2i(10,10);

```

```
glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE, data);
```

其中的 `width`、`height` 和 `data` 是 `readbmp` 函数中读出的图像数据，只要将这三个变量定义为全局变量，即可在这两个函数中都能访问到，完成的程序见随书的代码文件。

9.7 const 指针

对于下面涉及指针定义和操作的语句：

```
int a=1;
int *pi;
pi=&a;
*pi=58;
```

可以看到，一个指针涉及两个变量：指针本身 `pi` 和指向的变量 `a`。修改这两个变量的对应操作为“`pi=&a;`”和“`*pi=58;`”。

1. 指向常量的指针

指向常量的指针(常量指针)是指在指针定义语句的类型前加 `const`，表示指向的对象是常量。格式如下：

```
const 指针类型 *指针变量名;
```

例如：

```
const int a=1;
int b=2;
const int * pi=&a; //指针类型前加 const
*pi=3;             //error: 不能修改指针指向的常量
pi=&b;             //ok: 指针值可以修改
a=3;              //error
b = 3;            //ok
```

在上面的例子中，由于 `a` 是常量，将 `a` 的地址赋给指向常量的指针 `pi`，使 `a` 的常量性有了保证。如果企图通过指针的操作“`*pi=3;`”修改 `a`，则会引起“不能修改常量对象”(Cannot modify a const object)的编译错误。

`const int*pi` 的意思只是说 `pi` 指向的对象将视作常量看待，并不是说指针 `pi` 本身是常量，因而指针的值是可以被修改的，可以将另一个常量地址赋给指针“`pi=&b;`”。但是由于其指向的对象始终是作为常量处理，在指针的值被修改后，仍不能进行 `*pi` 的赋值操作，从而保护了被指向的常量不被修改。图 9-10 是指向常量的指针，其中阴影部分表示不能被修改。

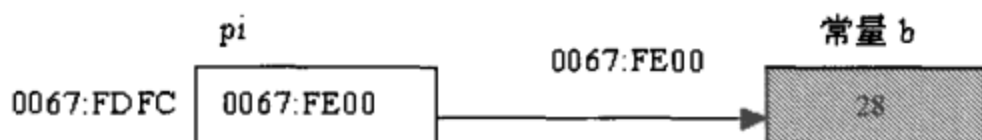


图 9-10 指向常量的指针

`const int *pi` 定义将指针指向的对象视作常量看待的另一层含义是：指针也可以指向变量，但在通过指针访问该变量时，该变量将被视为常量，值将不能被改变。故可以将一个变量地址赋给指针“`pi=&c;`”，这时，由于不能进行“`*pi=88;`”的赋值操作，从而保护了被指向的变量在指针操作中不被修改。定义指向常量的指针只限制指针的间接访问操作，而不能规定指针指向的值本身的操作规定性。例如，变量 `c` 可以修改，这在函数传递中经常被使用。

例如，下面的程序将两个一样大小的数组传递给一个函数，让其完成复制字符串的工作，为了防止作为源数据的数组遭到破坏，声明该形参为常量字符串。

```

#include <iostream >
using namespace std;
void mystrcpy(char* dest, const char* source)
{
    while(*dest++ = *source++);
}
void main()
{
    char a[20]="hello!";
    char b[20];
    mystrcpy(b,a);
    cout <<b <<endl;
}
  
```

程序运行结果如下：

hello!

变量字符串 `a` 传递给函数 `mystrcpy()` 中的 `source`，使之成为常量，不允许进行任何修改。但在主函数中，`a` 却是一个普通数组，没有任何约束，可以被修改。

由于数组 `a` 不能直接赋值给 `b`，所以要通过一个函数实现将数组 `a` 的内容复制给数组 `b`。

函数 `mystrcpy()` 中的语句是一个空循环，条件表达式是一个赋值语句。随着一个赋值动作，便将 `a` 数组中的字符赋给了 `b` 数组中对应的元素，同时两个数组参数都进行增量操作以指向下一个元素。只要所赋的字符值不是 ‘\0’ (条件表达式取假值)，则循环就一直进行下去。

常量指针定义“`const int *pi=&a;`”告诉编译器“`*pi`是常量，不能将`*pi`作为左值进行操作”。



注意：声明指向常量的指针格式有两种。

① `const` 指针类型 *指针变量名；

② 指针类型 `const` *指针变量名；

但是，第二种方式很容易与指针常量混淆，因此不提倡使用。

2. 指针常量

指针常量是指在指针定义语句的指针名前加 `const`，表示指针本身是常量。格式如下：

指针类型 * `const` 指针常量名 = 常量表达式；

例如：

```
char *s = "abc";
char *const pc = "abcd"; // 指针名前加 const 定义指针常量
pc = s;                  // error: 指针常量不能改变其指针值
*pc = 'b';               // ok: pc 内容为 'bbcd'
*pc++ = 'y';             // error: 指针常量不能改变其指针值
const int b = 28;
int *const pi = &b;      // error: 类型不一致
```

`pc` 是指针常量，在定义指针常量时必须初始化，就像常量初始化一样。这里初始化的值是字符串常量的地址，如图 9-11 所示。



图 9-11 指向变量的指针常量图

由于 `pc` 是指针常量，所以不能修改该指针值。“`pc=s;`”将引起一个“不能修改常量对象”(Cannot modify a const object)的编译错误。

`pc` 所指向的地址中存放的值并不受指针常量的约束，即 `*pc` 不是常量，所以“`*pc='b';`”的赋值操作是允许的。但“`*pc++=y;`”是不允许的，因为该语句修改 `*pc` 的同时也修改了指针的值。

由于此处 `*pi` 是不受约束的，所以，将一个常量的地址赋给该指针“`int* const pi=&b;`”是非法的，它将导致一个不能将“`const int *`”转换成“`int *`”的编译错误，因为那样将使修改常量(如“`*pi=38;`”)合法化。

注：不要把指针常量的定义格式：

指针类型 *const 指针常量名；

与指向常量的指针变量的第二种定义格式相混淆。

前者中的*const 一起说明后面定义的为指针常量，而 int const 则是说明指向的数据作为常量处理。

3. 指向常量的指针常量

将前两种情况综合一下，可以定义一个指向常量的指针常量(常量指针常量)，它必须在定义时进行初始化。格式如下：

const 指针类型 *const 指针常量名 = 常量表达式；

例如：

```
const int c=1;
int a;
const int * const cpc=&c;    //指向常量的指针常量
const int * const cpa=&a;    //ok
cpa=&c;                      //error: 指针值不能修改
*cpa=2;                      //error: 不能修改所指向的对象
a=2;                          //ok
```

cpc 和 cpa 都是指向常量的指针常量，它们既不允许修改指针值，也不允许修改所指的对象的值，见图 9-12。



图 9-12 指向常量的指针常量

如果初始化的值是变量地址(如&a)，那么不能通过该指针来修改该变量的值。也即“*cpa=39;”是错误的，将引起“不能修改常量对象”(Cannot modify a const object)的编译错误。但“a=39;”是合法的。

常量指针常量定义“const int * const cpc=&b;”告诉编译器：“cpc 和* cpc 都是常量，它们都不能作为左值进行操作”。

9.8 指针作为函数参数

设计一个函数，完成这样的任务：假设有 a、b 两个变量，通过调用函数，并以 a、b 做参数，在函数执行结束后，使 a、b 的值互换。读者的第一反应，函数应该如下：

范例 1:

```
void func(int x,int y)
{
    int t;
    t = x;
    x = y;
    y = y;
}
void main()
{
    int a,b;
    cin>>a>>b;
    func(a,b);
    cout<<"a="<<a<<"b="<<b<<endl;
}
```

那么这个程序执行的结果如何呢？

程序运行情况如下：

```
输入  3  4
输出  3  4
```

函数没能达到交换变量值目的的根本原因是，形参和实参的关系为：函数在进行调用时，将实参的值传递给形参，之后两者没有联系了，因此函数内部对 x、y 的交换不会影响到 a、b。那么该如何设计该函数呢？

函数的参数不仅可以是整型、实型和字符型等数据，还可以是指针类型。它的作用是将一个变量的地址传送到另一个函数中。

首先来看如下程序：

范例 2:

```
void func(int *x)
{
    *x = 10;
}
```

```

void main()
{
    int a = 0 ;
    int *p = &a;
    func(p);
    cout<<"a = "<<a<<endl;
}

```

程序运行结果如下：

a=10

这和范例 1 有什么不同呢？仔细观察可发现，函数的参数类型不一致，范例 2 的参数类型是指针。为了弄清问题，首先查看一下两个范例的内存使用情况，如图 9-13、图 9-14 所示。

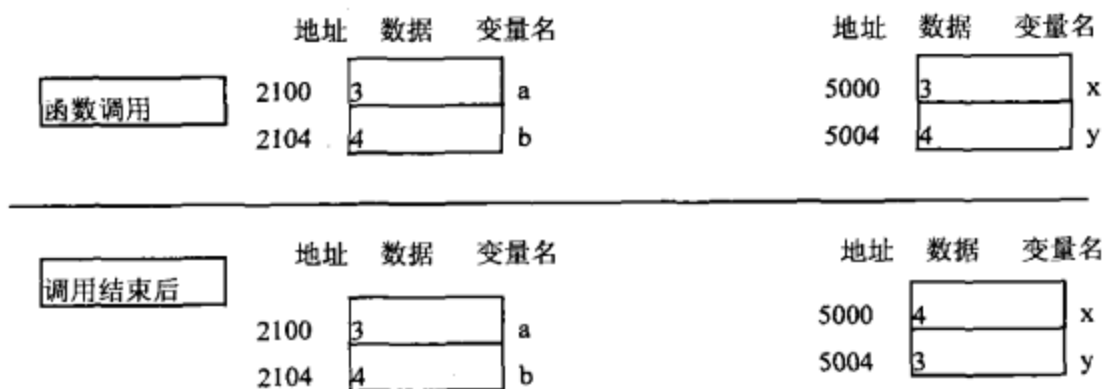


图 9-13 范例 1 的内存使用情况

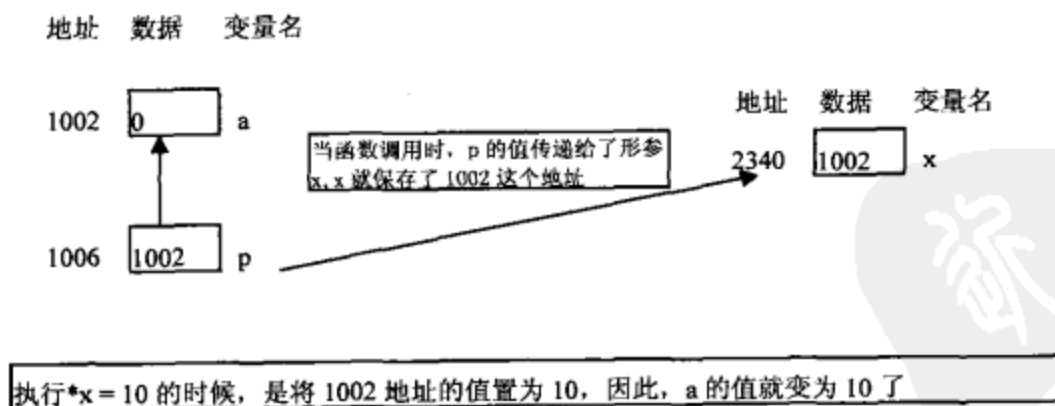


图 9-14 范例 2 的内存使用情况

通过分析图 9-13、图 9-14 内存的使用情况，可得出结论：如果函数的参数是指针，则在函数内部，可以通过指针访问到实参。那么范例 1 不能完成的任务就可做如下设计：

范例 3：

这次用指针类型的数据作函数参数。程序如下：

```
int func(int *p1,int*p2)
{
    int p;
    p=*p1;
    *p1=*p2;
    *p2=p;
}
void main(void)
{
    int a,b;
    int *pa,*pb;
    cin>>a>>b;
    pa=&a; pb=&b;
    func(pa,pb);
    cout<<"a = "<<a<<"b = "<<b<<endl;
}
```

程序运行情况如下:

输入:

5 9

输出:

a=9 b=5

对程序的说明:

func()是用户定义的函数,它的作用是交换两个变量(a和b)的值。func函数的两个形参p1、p2是指针变量。程序开始执行时,先输入a和b的值(输入5和9),然后将a和b的地址分别赋给指针变量pa和pb,即令pa指向a, pb指向b,接着执行函数func()。注意实参pa和pb是指针变量,在函数调用开始时,实参变量将它的值传送给形参变量,采取的依然是“值传递”方式。因此虚实结合后形参p1的值为&a。这时p1和pa都指向变量a, p2和pb都指向变量b。接着执行func函数的函数体,使*p1和*p2的值互换,也就是使a和b的值互换。函数调用结束后, p1和p2不复存在(已释放)。最后在main函数中输出的a和b的值已是经过交换的值(a=9, b=5)。

请注意交换*p1和*p2的值是如何实现的。如果写成以下这样就有问题了:

```
int swap(int *p1,int*p2)
{
    int *p;
    *p=*p1; //此语句有问题
    *p1=*p2;
    *p2=*p;
}
```


*p1 就是 a，是整型变量；而 *p 是指针变量 p 所指向的变量，但 p 中并无确定地址，用 *p 可能会造成破坏系统正常工作的状况。应该将 *p1 的值赋给一个整型变量，如程序所示那样，用整型变量 p 作为过渡变量实现 *p1 和 *p2 的交换。

注意，本例采取的方法是：交换 a 和 b 的值，而 p1 和 p2 的值不变。这和范例 1 是不一样的。可以看到，在执行 func 函数后，变量 a 和 b 的值交换了。这个改变不是通过形参值传回实参来实现的。

为了使在函数中改变了的变量值能被 main 函数所用，不能采取把要改变值的变量作为参数的办法，而应该用指针变量作为函数参数，在函数执行过程中使指针变量所指向的变量值发生变化。函数调用结束后，这些变量值的变化依然保留下来，这样就实现了“调用函数改变变量的值，在主调用函数(如 main 函数)中使用这些改变了的值”的目的。

如果想通过函数调用得到 n 个要改变的值，可以按以下思路实现。

- ① 在主调函数中设 n 个变量，用 n 个指针变量指向它们。
- ② 然后将指针变量作实参，将这 n 个变量的地址传给所调用的函数的形参。
- ③ 通过形参指针变量，改变该 n 个变量的值。
- ④ 主调函数中就可以使用这些改变了值的变量。

请读者按此思路仔细理解范例 3 的程序。

请注意，不能企图通过改变指针形参的值而使指针实参的值也改变。请看下面的程序：

```
int func(int *p1,int*p2)
{
    int *p;
    p=p1;
    p1=p2;
    p2=p;
}
void main(void)
{
    int a,b;
    int *pa,*pb;
    cin>>a>>b;
    pa=&a; pb=&b;
    if(a<b)
        func(pa,pb);
    cout<<"a = "<<*pa<<"b = "<<*pb<<endl;
}
```

程序的意图是：交换 a 和 b 的值，使 a 保存大值。其设想是：

- ① 先使 pa 指向 a, pb 指向 b。
- ② 调用 func 函数, 将 pa 的值传给 p1, pb 的值传给 p2。
- ③ 在 func 函数中使 p1 与 p2 的值交换。
- ④ 形参 p1、p2 将地址传回实参 pa 和 pb, 使 pa 指向 b, pb 指向 a。然后输出 *pa、*pb, 想得到输出 “9, 5”。

但是这是办不到的, 程序实际输出为 “5, 9”。问题出在第④步。C 语言中实参变量和形参变量之间的数据传递是单向的“值传递”方式。指针变量作函数参数也要遵循这一规则。调用函数不能改变实参指针变量的值, 但可以改变实参指针变量所指变量的值。函数的调用可以(而且只可以)得到一个返回值(即函数值), 而运用指针变量作参数, 可以得到多个变化了的值。如果不用指针变量是难以做到这一点的。

以下示例的功能是:

输入 a、b、c 三个整数, 按大小顺序输出。

```
int swap(int *p1,int*p2)
{
    int p;
    p=*p1;
    *p1=*p2;
    *p2=p;
}
int exchange(int *q1,int *q2,int *q3)
{
    if(*q1<*q2)
        swap(q1,q2);
    if(*q1<*q3)
        swap(q1,q3);
    if(*q2<*q3)
        swap(q2,q3);
}
void main(void)
{
    int a,b,c,*p1,*p2,*p3;
    cin>>a>>b>>c;
    p1=&a,p2=&b,p3=&c;
    exchange(p1,p2,p3);
    cout<<"a = "<<a<<"b = "<<b<<"c = "<<c<<endl;
}
```

程序运行情况如下:

输入:

9 0 10

输出:

a=10 b=9 c=0

9.9 指针函数

返回指针的函数称为指针函数。一般形式为:

类型标识符 * 函数名(参数表)

例如, `int * a (int x, int y)` 声明一个函数, 函数名为 `a`, 其返回值类型是“指向整型的指针”, 函数形式参数为 `int x` 和 `int y`。

以下示例的功能是, 有若干学生的成绩(每个学生四门课程), 要求用户在输入学生序号(从 0 开始)后, 能输出该学生的全部成绩。

分析: 如图 9-15 所示, 设计一个指针 `pscore` 指向一个学生的四门成绩, 语句如下:

```
float (*pscore)[3]
```

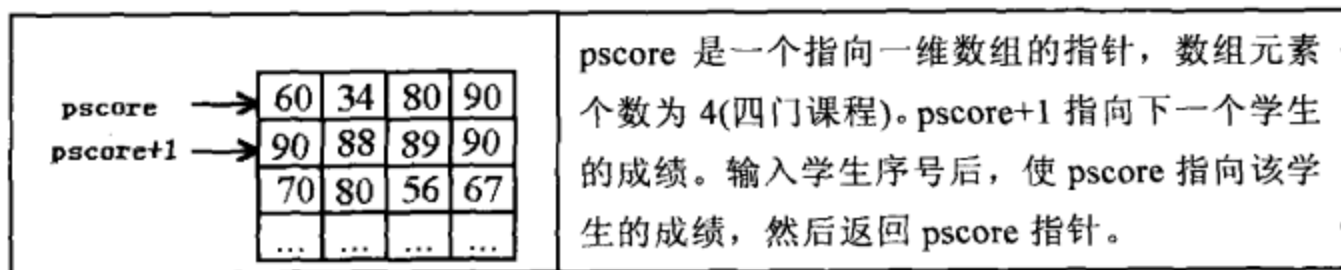


图 9-15 程序分析示意(1)

程序如下:

```
float * search( float (*pscore)[3], int n) ;
void main()
{
    static float score[][4] = {{60,34,80,90},{90,88,89,90},{70,80,56,67}};
    float *p;
    int i, m;
    cout<<"请输入学号:"<<endl;
    cin>>m;
    printf("该生的成绩是:\n", m);
    p = search(score, m); //在 score 数组中查询 m 号学生的成绩,
                          //查询结果为指向成绩的指针
    for(i=0; i<4; i++)
        cout<<*(p+i);
}
float * search( float (*pscore)[3], int n)
```

```

{
    float *pt;          //pt 是指向实数的指针, pscore 是指向数组的指针
    pt = *(pscore+n);   //pt = (float *) (pscore + n)
    return pt;
}

```

对上例中的学生, 找出有不及格成绩的学生及其学号。

分析: 上例中, `search` 函数返回学生成绩的首地址。本例用返回地址来区分学生成绩中有无不及格课程, 若有不及格课程, 仍返回学生成绩的首地址; 若无不及格课程, 则返回学生成绩的末地址(等于下一个学生的首地址), 如图 9-16 所示。

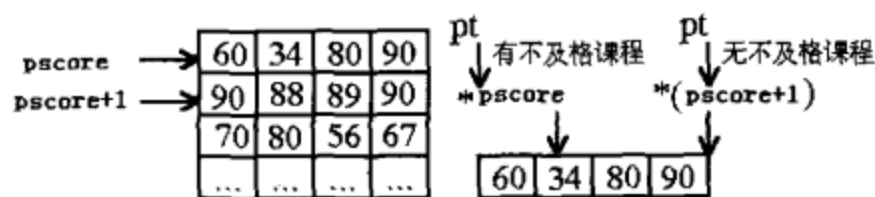


图 9-16 程序分析示意(2)

程序如下:

```

float * search( float (*pscore)[4] );
void main()
{
    static float score[][4] = {{60,34,80,90},{90,88,89,90},{70,80,56,67}};
    float *p;
    int i, j;
    for(i=0; i<3; i++)          //三个学生
    {
        p = search(score+i);
        if (p == *(score+i))    //p 指向 i 号学生成绩首地址, 该生有不及格课程
        {
            cout<<"成绩:\t", i); //显示该生的四门课程成绩
            for(j=0; j<4; j++)
            {
                cout<<*(p+j)<<" ";
            }
            cout<<endl;
        }
    }
}

float * search( float (*pscore)[4] )
{
    int i;
    float *pt;

```

```

pt = *( pscore + 1); /*先设 pt 指向成绩末地址(等于下一个学生成绩首地址),
                        即先假设无不及格成绩 */
for(i=0; i<4; i++) /* 查四门课程中有无不及格成绩 */
{
    if ( *(* pscore +i)<60 ) pt = * pscore; /*若有不及格课程, 则 pt 指向成绩
                                                首地址*/
    return pt;
}
}

```

指针函数不能把在它内部说明的具有局部作用域的数据地址作为返回值。

下面的程序示例打印一个用整型指针指向的整数值。

```

#include <iostream.h>
int* getInt(char* str) //指针函数
{
    int value=20;
    cout <<str <<endl;
    return &value; //warning: 将局部变量的地址返回是不妥的
}
void somefn(char* str)
{
    int a=40;
    cout <<str <<endl;
}
void main()
{
    int* pr=getInt("input a value:");
    //赋值取自返回的指针值
    cout <<*pr <<endl; //第一次输出*pr
    somefn("test2.");
    cout <<*pr <<endl; //第二次输出*pr
}

```

程序运行情况如下:

```

input a value:
20
test2.
4435500

```

9.10 函数指针

每个函数在编译后其函数名对应于该函数执行代码的入口地址。对一个函数只能做两

件事：调用一个函数或取一个函数的入口地址。和数组名类似，函数名代表函数的入口地址，通过取地址运算符“&”也可以得到函数的入口地址。由于函数名不是变量，所以函数不能作为数组元素，也不能作为其他函数的形式参数。那么如果确实想做一个函数列表，其中存放的是各种函数，当需要时直接在列表内找相应的函数并传递给主调函数使用时，该怎么做呢？C++通过定义函数指针的方式来解决这类问题。函数指针即指向函数的指针，它是一种特殊的指针变量，这种变量专门用来存放同种类型函数的入口地址。函数指针既可以作为普通函数的替代形式调用函数体完成函数功能，又可以作为函数的参数传递，还可以作为数组元素，构造一个指向函数的指针数组。

指向函数的指针变量一般定义方式是：

数据类型标识符 (*指针变量名) (参数表)

这里的“数据类型标识符”是指函数返回值的类型。

函数指针通过赋值，可以存储一个函数的入口地址。例如：

`void scopy(char *q, char *p)`是一个函数，函数名 `scopy` 代表函数的入口地址，或者说是一个指向该函数的指针，这个指针的类型表示为下列这样一个指向函数的指针类型：
`void(*) (char *q, char *p);`

下式命名一个指向函数的指针变量 `pfv`，并由函数 `scopy` 初始化：

`void(*pfv)(char *q, char *p)=scopy;`

指向函数的指针变量可以用函数名赋值，或者通过取地址运算符“&”得到函数的地址后赋值：

```
pfv=scopy;
pfv=&scopy();
```

`pfv` 经初始化(或赋值)后，将存储 `scopy` 函数的入口地址。

由于一个函数不能直接以函数作为参数，所以当需要一个函数需要函数参数时必须借用函数指针。

下面先通过梯形法求积分的例子介绍指向函数的指针作为参数的必要性。

梯形法求积分是一种求函数定积分的近似方法。对函数 $f(x)$ 将积分区间 $[a, b]$ 分成 n 份，每一份看作一个近似梯形，函数在该区间的定积分就是所有近似梯形的面积和。对不同的积分函数仅是函数关系式 $f(x)$ 不同，积分方法都是一样的。设积分步长为 $step=(b-a)/n$ ，则函数 $f(x)$ 在区间 $[a, b]$ 定积分(即面积)为：

$$s=step*(f(x_0)+f(x_1))/2+step*(f(x_1)+f(x_2))/2+...+step*((f(x_{n-1})+f(x_n))/2)$$

$$= \text{step} * (f(X_0)/2 + f(X_1) + f(X_2) + \dots + f(X_{n-1}) + f(X_n)/2)$$

函数 `integer()` 是一个用梯形法求任一函数定积分的通用函数，不同的函数有不同的解析式，由解析式决定自变量在每一个小积分区间端点处的函数值。函数 `integer()` 以一个指向函数的指针为参数，由该参数调用欲求定积分的函数，另两个参数是积分区间。

```
//文件名: INTEGER. CPP
//利用指向函数的指针求解定积分
#include <stdio.h>
#include <iomanip.h>
double f1(double x)           //y=1+2x2
{ return(1+2*x*x); }
double f2(double x)           //y=1+2x2+3x3
{ return(1+2*x*x+3*x*x*x); }
double f3(double x)           //y=1+2x2+4x4
{ return(1+2*x*x+4*x*x*x*x); }
double integer(double(*func)(double), float, float);
void main()
{
    double fixint1, fixint2, fixint3;
    fixint1=integer(f1, 0.0, 1.0);
    fixint2=integer(f2, 0.0, 2.0);
    fixint3=integer(f3, 0.0, 3.0);
    printf("%f %f %f\n", fixint1, fixint2, fixint3);
    return;
}
double integer(double(*func)(double), float a, float b)
{
    double result, step;
    result=((*func)(a)+(*func)(b))/2;
    step=(b-a)/100;
    for(int i=1; i<100; i++)
        result+=(*func)(a+i*step);
    result*=step;
    return(result);
}
```

在上例的 `integer()` 函数中，第一个参数 `double(*func)(double)` 说明是一个指向函数的指针，所指向的函数以 `double` 型为形式参数，而且该函数的返回类型为 `double`。特别要注意的是在 `(*func)` 两边的括号，由于 `()` 的优先级高于 `*`，如果忽略了括号而写成 `double *func(double)`，则表示是一个函数，该函数以 `double` 型为形式参数，该函数的返回值为一个指向 `double` 型的指针。

在理解函数与函数指针时应注意以下几点：

- 函数的调用可以通过函数名调用,也可以通过函数指针调用(即用指向函数的指针变量调用)。
- `(*p)()`表示定义一个指向函数的指针变量,它不是固定指向哪一个函数的,而只是表示定义了这样一个类型的变量,它是专门用来存放函数的入口地址的。
- 在给函数指针变量赋值时,只需给出函数名而不必给出参数。`p=max`;因为是将函数入口地址赋给 `p`,而不涉及实参与形象的结合问题。
- 用函数指针变量调用函数时,只需将`(*p)`代替函数名即可(`p`为指针变量名),在`(*p)`之后的括号中根据需要写上实参。
- 对指向函数的指针变量,其常用的用途之一是把指针作为参数传递到其他函数。

例: 设一个函数 `process`, 在调用它的时候, 每次实现不同的功能。输入 `a` 和 `b` 两个数, 第一次调用 `process` 时找出 `a` 和 `b` 中的大者, 第二次找出其中的小者, 第三次求 `a` 与 `b` 之和。

程序如下:

```
main( )
{
    int max(int,int),min(int,int),add(int,int);
    int a, b;
    cout<<"请输入 a 和 b 的值:"
    cin>>a>>b;
    cout<<"max = ";
    process(a,b,max);
    cout<<"min = ";
    process(a,b,min);
    cout<<"sum = ";
    process(a,b,add);
}
int max(int x,int y)
{
    int z;
    if(x>y)
        z = x;
    else
        z = y;
    return(z);
}
int min(int x,int y)
{
    int z;
    if (x<y)
        z = x;
    else
        z = y;
```




```

        return(z);
    }
    int add(int x,int y)
    {
        int z;
        z=x+y;
        return(z);
    }
    int process(int x,int y, int(*fun)(int,int))
    {
        int result;
        result=(*fun)(x,y);
        cout<<result<<endl;
    }

```

运行情况如下:

```

请输入 a 和 b 的值:2,6
max=6
min=2
sum=8

```

max、min 和 add 是已定义的三个函数,分别用来实现求大数、求小数和求和功能。在 main 函数中第一次调用 process 函数时,除了将 a 和 b 作为实参将两个数传给 process 的形参 x, y 外,还将函数名 max 作为实参将其入口地址传送给 process 函数中的形参——指向函数的指针变量 run,这时,process 函数中的(*fun)(x,y)相当于 max(x,y),执行 process 可以输出 a 和 b 中的大者。在 main 函数中第二次调用 process 函数时,改以函数名 min 作实参,此时 process 函数的形参 fun 指向函数 min,在 process 函数中的函数调用(*fun)(x,y)相当于 min(x,y)。同理,第三次调用 process 函数时,(*fun)(x,y)相当于 sum(x,y)。

函数指针用途广泛,在 9.6.2 节中,OpenGL 应用程序就是通过函数指针告诉系统,到底用哪一个函数来执行窗口变换,当需要显示的时候,系统又该调用哪个函数。

9.11 指针数组

由若干个同类型指针变量组成的数组称为指针数组,其中每一个元素都是一个指针变量。指针数组的一般形式为:

类型标识符 *数组名 [数组元素个数]

例如, int * p[4];定义一个指针数组,数组名 p,有 4 个元素,每一个元素是指向整型变量的指针。



注意: `int (*p)[4]`与上面的例子不同, 它定义了一个指针变量, 指向有 4 个元素的一维数组。

1. 指针数组一般用法

指针数组与普通的数组并没有什么不同, 只是其中的元素都是指针而已, 对数组的各种操作对于指针数组同样适用。同理, 指针数组中的每个指针元素, 对于指针的各种操作也同样适用。

指针数组多用于处理多个字符串或建立索引表上。

字符串本身是一维数组, 多个字符串可以用二维数组来处理, 但会浪费许多内存。用指针数组处理多个字符串, 则不会浪费内存, 如图 9-17 所示。

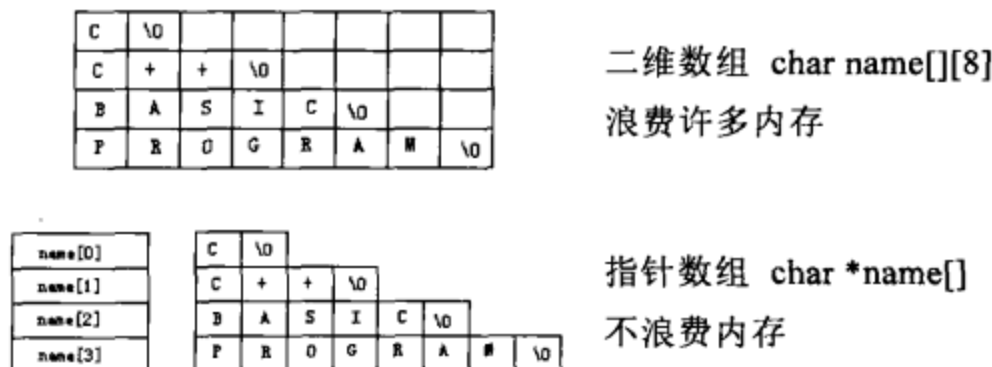


图 9-17 二维数组与指针数组所占内存的比较

由图 9-17 可知, 指针数组的使用不仅使原有的结构没有改变, 而且在存储上更加节省了数据的存储空间, 使程序变得更加灵活。

以下示例能将若干字符串按字母顺序(由小到大)输出。

```
#include "stdio.h"
#include "string.h"
void sort(char *name[], int n);    //排序函数原型
void print(char *name[], int n);   //输出函数原型
void main()
{
    static char *name[] =
        {"c", "c++", "basic", "program"};
    int n = 4;
    sort(name, n);    //排序
                      //输出

    int i;
    for (i=0; i<n; i++)
        printf("%s\n", name[i]);
}
```

```

void sort(char *name[], int n) //冒泡法排序
{
    char *temp;
    int i, j, k;
    for(i=0; i<n-1; i++)          //n个字符串，外循环n-1次
    {
        k = i;
        for(j=i+1; j<n; j++)      //内循环
            if (strcmp(name[k], name[j]) > 0) k = j;
        //比较 name[k] 与 name[j] 的大小，较小字符串的序号保留在 k 中
        if (k != i)
        { //交换 name[i] 与 name[k] 的指向
            temp = name[i];
            name[i] = name[k];
            name[k] = temp;
        }
    }
}

```

2. main 函数的参数

利用命令行方式启动程序的时候，运行程序的命令行中，可以包含参数，例如：

命令名 参数1 参数2 ... 参数n

“命令名”是可执行文件，例如 format(格式化磁盘的一个命令)，执行该命令时包含两个字符串参数：“format.exe”和“a:”。

如果编写一个 format 程序，在源程序 format.c 中，用 main() 函数的参数来接收命令的参数，格式如下。

```
main(int argc, char *argv[])
```

其中，argc 表示命令行参数的个数(包括命令名)，指针数组 argv 用于存放参数(包括命令名)，上例中：argc = 2; argv[0] = "format.exe", argv[1] = "a:"；如图 9-18 所示。

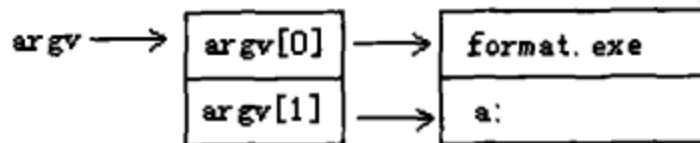


图 9-18 argv 数据表

以下示例中设文件名是 format 代码如下:

```
#include <iostream>
using namespace std;
main(int argc, char *argv[])
{
    while(argc>1)
    {
        cout<<(*argv)<<endl;
        argv++;
        argc--;
    }
}
```

程序运行情况如下:

输入:

format.exe a:

输出:

format.exe

a:

该程序可改写为:

```
main(int argc, char **argv[])
{
    while( argc-- > 0)
        cout<<(*argv++)<<endl;
}
```

例如, echo(参数回送)命令的程序代码如下:

```
main(int argc, char *argv[])
{
    while(--argc>0)
    {
        cout<<(*++argv);
        (argc>1)? {cout<<' '}:{cout<<endl};
    }
}
```

或写为:

```
main(int argc, char *argv[])
{
    int i;
    for(i=1; i<argc; i++)
    {
```



```

    cout<<argv[i];
    (i<argc-1)? {cout<<' '}:{cout<<endl};
}
}

```

9.12 指向指针的指针

指针变量作为变量的一种本身也有一个存储单元，故它本身也有一个存储地址，而该地址又可以赋给另一个指针变量，这后一个指针变量就叫做指向指针的指针。

如图 9-19 所示，变量 *a* 的空间中存放的是 *a* 的值，变量 *p* 为一个指针变量，其中存放的是变量 *a* 的地址，而变量 *pp* 则是一个指向指针 *p* 的指针变量。

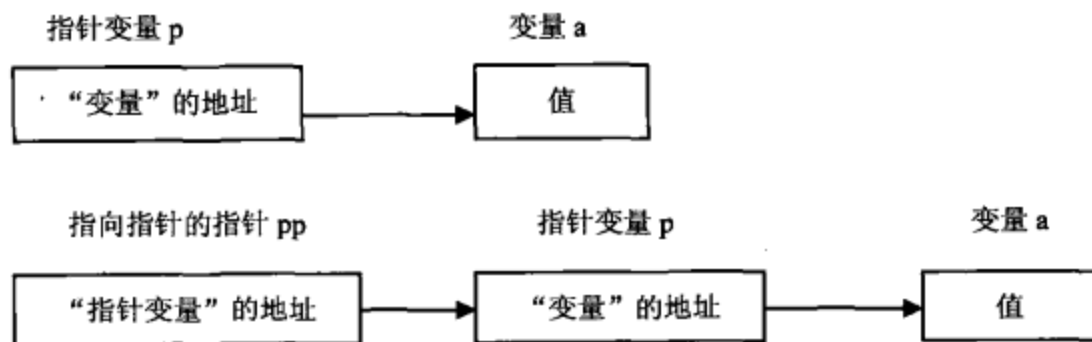


图 9-19 指针于变量的关系图

指向指针的指针变量的定义格式为：

数据类型标识符 ** 变量名；

例如：

```
int ** pp;
```

对应于上面的图，定义各变量为：

```

int a = 55;
int *p = &a;
int **pp = &p;

```

指向指针的指针类似于指针数组中的数组名，通过指向指针的指针可以得到下一级指针的地址，从而获得其中存储的变量的地址。也可以直接将指针数组的地址赋值给指向指针的指针。

使用举例，如图 9-20 所示。

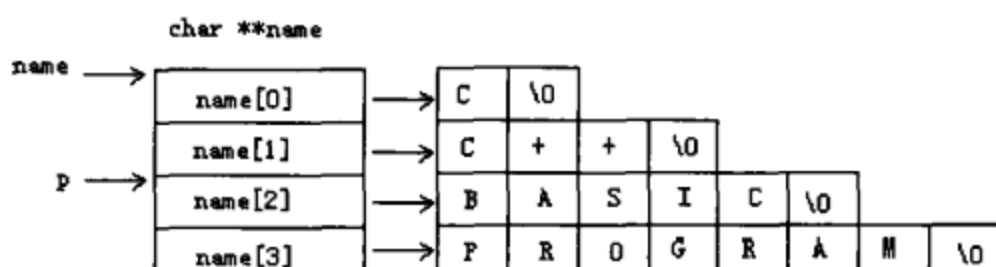


图 9-20 指向指针的指针使用举例

设字符指针数组 `char **name` 指向字符串。定义指向指针的指针 `p`: `char **p`, 使其指向 `name`。例如:

```
p = name + 2;
cout<<(*p)<<endl; //输出 name[2] 的值, 它是字符串 "BASIC" 的地址
cout<<(**p)<<endl; //输出 name[2] 指向的字符串
```

此示例的程序如下:

```
void main()
{
    static char *name[] = {"C", "C++", "BASIC", "PROGRAM"};
    char **p;
    int i;
    for(i=0; i<4; i++)
    {
        p = name + i;
        cout<< i<< ' ' << (*p) << ' ' << (**p) << endl;
    }
}
```

运行程序后输出:

```
0 100 C
1002 C++
0106 BASIC
0112 PROGRAM
```

通过上例可以看到, 指向指针的指针是二级指针, 同理, 当再定义指向这种变量的指针时, 即构成多级指针。

多级指针的定义格式如下, 其示意图如图 9-21 所示。

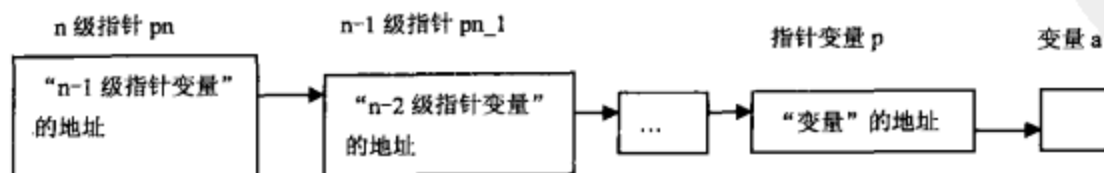


图 9-21 多级指针

z 数据类型标识符 *...* n 级指针变量名
n 个*

如三级指针，其定义格式为：

```
int *** p;
```

由于级数过多会使跟踪困难，一般在程序中很少使用超过二级的指针。

无论是普通指针，还是多级指针，它们能运用*操作符的次数是受其级数限制的，每个指针变量都只能最多进行其定义级数 n 次*操作。

即对于一般的指针(如 `int *p`)来说，可以对它进行的操作有：

- ① `p=&a;` //为指针赋值
- ② `*p=55;` //通过指针操作指向的变量

而对于二级指针，如 `int **p2`，可以对它进行的操作有：

- ① `int **p2;`
`int a;`
`int *p1=&a;`
`p2 = &p1;` //为指针赋值
- ② `int a,b;`
`int *p1=&a;`
`int **p2 = &p1;`
`*p2 = &b;` //操作其指向的指针变量的值
- ③ `int a,b;`
`int *p1=&a;`
`int **p2 = &p1;`
`**p2 = 55;` //操作其指向的指针指向的变量的值

以此类推，对于 n 级指针，可以进行的操作共有 n+1 中，其中 n 次为*操作。

因此，在需要指向指针的指针时不能用普通的指针变量来将其代替。请看下面的例子：

```
int a=3;
int *p=&a;
int *pp=&p;
```

最后一句的 `int *pp=&p` 是给一个指针赋初值，不过这个初值是一个“指针的地址”，而不是通常的“变量地址”(所以此时编译器会有警告)，但是还是可以通过的。

初一看，这不就是将 `pp` 指向指针变量 `p` 了吗？和 `int **p=&p` 应该没有什么区别吧。可是，事实并非如此，这个 `pp` 指针没有任何用处，因为 `pp` 指向指针 `p` 的地址，也就是 `&p`(即 `pp=&p`)；那么 `*pp=&a`，即 `*pp` 表示变量 `a` 的地址(不能说 `pp` 指向变量 `a`)。

如此看来,要操作变量 `a` 的话,可以使用 `*p` 来操作(如 `*p=8`),但是通过 `pp` 则不行,因为 `*pp` 只是表示变量 `a` 的地址,并不能操作该地址中的内容,要在地址前加 `*` 来表示操作地址内的内容,但这是编译器所不允许的,因为在定义的时候, `pp` 只是个一级指针,它只能运用一次 `*` 操作符。

这种情况看上去就像是,看得到却摸不到(不能操作变量)。

所以如上定义一个指针来指向一个指针,是没有多大用处的,它唯一可以改变的就是指针 `pp` 的指向(如 `*pp1=1000`,则指针 `pp` 指向的地址变为 1000),但这种改变是危险的和致命的,也是没有什么实际用处的,因为 `pp` 指向的改变是很容易的(根本没有必要通过定义一个指向自己的指针来改变)。例如:

```
int a=3,b;  
int *p=&a;  
int *p1=&b;  
p=p1;
```

所以,如果要在开头定义一个指针来指向指针,正确方法应该如下:

```
int a=3;  
int *p=&a;  
int **pp=&p;
```

`pp` 是个指针的指针,例如 `**pp=8`,则 `a=8`;等同于 `*p=8`。

9.13 常见的内存错误及其对策

发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误,通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状,时隐时现,增加了改错的难度。

常见的内存错误及其对策如下。

(1) 内存分配未成功,却使用了它。

编程初学者常犯这种错误,因为没有意识到内存分配会不成功。常用解决办法是,在使用内存之前检查指针是否为 `NULL`。如果指针 `p` 是函数的参数,那么在函数的入口处用 `assert(p!=NULL)` 进行检查。如果是用 `malloc()` 函数或 `new` 操作符来申请内存,应该用 `if(p==NULL)` 或 `if(p!=NULL)` 进行防错处理。

(2) 内存分配虽然成功,但是尚未初始化就引用它。

犯这种错误主要有两个起因:一是没有初始化的观念;二是误以为内存的缺省初值全为零,导致引用初值错误(例如数组)。

内存的默认初值究竟是什么并没有统一的标准, 尽管有些时候为零值, 宁可信其无不可信其有。所以无论用何种方式创建数组, 都别忘了赋初值, 即便是赋零值也不可省略, 不要嫌麻烦。

(3) 内存分配成功并且已经初始化, 但操作越过了内存的边界。

例如在使用数组时经常发生下标“多1”或者“少1”的操作。特别是在 for 循环语句中, 循环次数很容易搞错, 导致数组操作越界。

(4) 忘记了释放内存, 造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足, 你看不到错误。终有一次程序突然死掉, 系统出现提示: 内存耗尽。

动态内存的申请与释放必须配对, 程序中 malloc 与 free 的使用次数一定要相同, 否则肯定有错误(new/delete 同理)。

(5) 释放了内存却继续使用它。

有下面三种情况。

① 程序中的对象调用关系过于复杂, 实在难以搞清楚某个对象究竟是否已经释放了内存, 此时应该重新设计数据结构, 从根本上解决对象管理的混乱局面。

② 函数的 return 语句写错了, 注意不要返回指向“栈内存”的“指针”或者“引用”, 因为该内存在函数体结束时被自动销毁。

③ 使用 free 或 delete 释放了内存后, 没有将指针设置为 NULL, 导致产生“野指针”。

9.14 引用的定义

引用是 C++ 引入的新语言特性。从语意上来说, 引用就是一个变量的别名, 就好像古代人的“字”和“号”, 东坡居士和苏轼只是一个人的不同称呼。

引用的声明格式为:

类型标识符 & 引用名 [= 别名所代表的对象];

其中: 类型标识符是引用的目标变量的类型, 引用名的命名规则可以为任意合法的变量名, 遵循 C++ 中标识符的命名规则。如果引用不是在参数表中定义的, 就必须初始化, 即必须在定义时就确定这个别名所代表的变量、函数等对象。从那时起, 引用作为目标的别名而使用, 对引用的改动实际就是对目标的改动。

例如:

```
int i = 0;
int & iRef = i;
```

在上面的例子中定义了一个对变量 *i* 的引用 *iRef*。以后，所有对引用的操作都是对变量 *i* 的操作，其产生的影响与对变量直接操作完全一样。例如：

```
iRef++;  
等价于：i++;
```

下面介绍什么能被引用。

声明为类型 *T* 的引用只能用来引用 *T* 类型的变量或对象，或至少被引用的变量能够转换成 *T* 类型。

请注意，下面的代码是错误的：

```
double& rr = 1 ;①
```

因为对于立即数 1 来说没有内存地址，因此 *rr* 不知道引用哪个变量的地址，即 *rr* 没有地址可以使用。

但是如下的写法是正确的：

```
const double& rr=1 ;②
```

因为在这种情况下：

- (1) 计算机首先做必要的类型转换。
- (2) 然后将结果置于临时变量。
- (3) 最后，把临时变量的地址作为初始化的值。

事实上，临时变量并不在存放局部变量的栈区。

同指针不同的是，同种类型的指针可以声明指针数组，但不能声明引用的数组，例如：

```
int a[10];  
int& ra[10]=a; //error
```

但是可以声明对数组元素的引用：

```
int &ra1=a[6]; //ra1 代表数组元素 a[6]
```

也可以声明对数组的引用：

```
int (&ra2)[10]=a; //ra2 代表数组 a，定义 ra2
```

请注意，引用的数组和对数组的引用不是一个概念。声明了对数组的引用以后，在使

① 参见 C++手册。

② 很多 C++教材前面都缺少了 `const`，这样编译器所做的工作并不一样，因此是不对的。

用每个数组元素时便可以把引用名当作数组名看待使用。

例如：

```
int a[10];
int (&ra2)[10]=a;      //ra2 代表数组 a，定义 ra2
ra[3] = 100;
```

以下示例将实现对数组和数组元素的引用。

```
#include <iostream>
using namespace std;
void main()
{
    int a[10],i;
    int (&ieref)[10]=a;      //定义对数组的引用
    int &r = a[5];          //定义对数组元素的引用
    for(i=0;i<10;i++)
    {
        iref[i] = i;      //通过引用对数组初始化
    }
    cout<<" Iref 内的数据"<<endl;
    for(i=0;i<10;i++)
    {
        cout<<ieref[i]<<' ';    //通过引用方式输出数组元素
    }
    cout<<endl;
    cout<<"data 内的数据"<<endl;
    for(i=0;i<10;i++)
    {
        cout<<a[i]<<' ';    //通过数组方式输出数组元素
    }
    cout<<endl;
    cout<<r<<endl;
    cout<<(&ieref)<<endl;
    cout<<a<<endl;
}
```

程序中定义了对数组 `a[10]` 的引用 `iref` 和对第五个数组元素的引用 `r`，然后通过数组的引用对数组进行了初始化，最后分别以引用和数组的方式将数组的元素进行了输出显示。

程序运行结果如下：

```
Iref 内的数据
0 1 2 3 4 5 6 7 8 9
data 内的数据
0 1 2 3 4 5 6 7 8 9
```

```
5
00100000
00100000
```

引用本身不是一种数据类型，所以没有引用的引用，也没有引用的指针。例如：

```
int a;
int& ra=a;
int& *p=&ra; //error: 企图定义一个引用的指针
```

引用不是类型，引用只有声明没有定义，声明引用在概念上不产生内存空间，所以，在引用之上的引用不存在，而且引用的指针指向谁呢？

但是，由于指针也是变量，所以可以有指针变量的引用：

```
int * a;
int* &p=a; //表示 int*的引用 p 初始化为 a
int b=8;
p=&b; //ok! p 是 a 的别名，是一个指针
```

以下示例中包含对指针变量的引用。程序代码如下：

```
#include <iostream>
using namespace std;
void main()
{
    int a = 15;
    int * p = &a;
    int * &ref = p;                                //定义指针的引用
    cout<<"a: "<<a<<endl;                          //输出 a 的值
    cout<<"*p: "<<(*p)<<endl;                        //输出 p 指向单元的值
    cout<<"*ref: "<<(*ref)<<endl;                    //输出*ref 的值
    cout<<"p: "<<p<<endl;                          //输出 p 的值
    cout<<"ref: "<<ref<<endl;                        //输出 ref 的值

    *ref = 37;
    cout<<"a: "<<a<<endl;
    cout<<"*p: "<<(*p)<<endl;
    cout<<"*ref: "<<(*ref)<<endl;                    //输出*ref 的值
    cout<<"p: "<<p<<endl;                          //输出 p 的值
    cout<<"ref: "<<ref<<endl;                        //输出 ref 的值

    cout<<"The address that p point to is "<<p<<endl;
    cout<<"The value of ref is "<<ref<<endl;
    cout<<"The address of pointer p is "<<&p<<endl;
    cout<<"The address of ref seen by us is "<<&ref<<endl;
```

```

}

```

程序运行结果如下：

```

a: 15
*p: 15
*ref: 15
p: 00100000
ref: 00100000
a: 37
*p: 37
*ref: 37
p: 00100000
ref: 00100000
The address that p point to is 00100000
The value of ref is 00100000
The address of pointer p is 00100008
The address of ref seen by us is 00100008

```

指针变量的引用，见图 9-22。

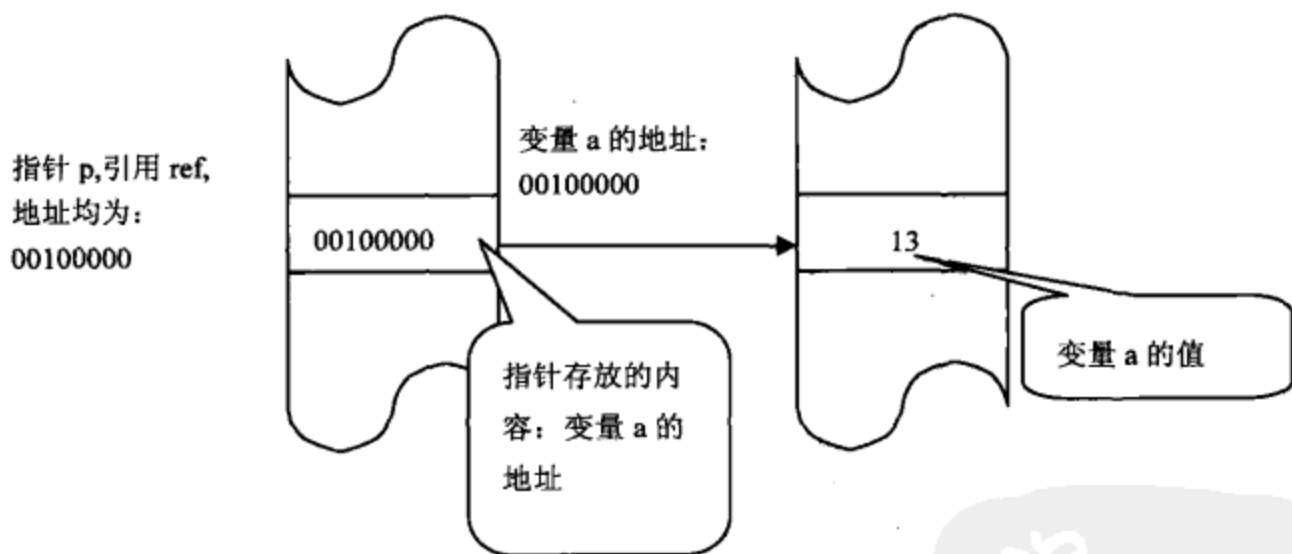


图 9-22 指针变量的引用

在定义引用时，还可以通过指针定义对指针指向变量的引用，如：

```

int a=10;
int *p=&a;
int &ref=*p;    //ref 代表 p 所指向的那个对象，即变量 a

```

请注意区分前面已经讲过的对指针的引用：`int * &ref=p;`对指针的引用在使用时和使用指针本身相同，在对指针指向的变量操作时需加*号；而对变量的引用则不需要。例如：

```

#include <iostream>
using namespace std
void main()

```

```

{
    int a=10;
    int *p=&a;
    int &ref=*p;           //ref 代表 p 所指向的那个对象，即变量 a
    int * &refp=p;         //refp 代表指针 p
    cout<<"ref: "<<ref<<endl;
    cout<<"*refp: "<<refp<<endl;
}

```

程序输出结果为：

```

ref: 10
*refp: 0012FF7C

```

对 **void** 进行引用是不允许的。例如：

```
void& a=3; //error
```

void 只是在语法上相当于一个类型，本质上不是类型，没有任何一个变量或对象其类型为 **void**。

引用不能用类型来初始化，例如：

```
int& ra=int; //error
```

因为引用是变量或对象的引用，而不是类型的引用。

有空指针，无空引用。不应有下面的引用声明，否则会有运行错误：

```
int& ri=NULL; //毫无意义
```

9.15 使用引用访问数据

引用在初始化以后就可以像普通变量一样使用了。下面是几个相关的例子。

以下示例用于计算圆的面积，其中半径通过引用来访问。

```

#include <iostream>
using namespace std;
void main()
{
    float radius=30,area;
    float &ref=radius;
    area = 3.14 * ref * ref;
    cout<<"The total area of the circle is "<<area<<endl;
}

```

程序运行结果如下:

The total area of the circle is 2826

在使用引用访问数据时要注意以下几点。

(1) 对数组的引用在访问数组元素时要加相应的下标, 对指针指向变量的访问要通过操作符*来实现。

以下示例可实现填数字游戏。

```
#include <iostream>
using namespace std;
int array[3][3]={
    6,7,2,
    1,5,9,
    8,3,4
};

void main()
{
    int answer;
    char *p = "Hey!Clever boy,can you help me complete the puzzle?";
    char *result[2] = {"Congratulations!You've done an excellent job!",
        "Oh,my god!You're wrong!"};
    int (&rarray)[3][3]=array;
    cout<<p<<endl;
    cout<<"Please find the rule in the numbers, and fill in the gap with an
appropriate number"<<endl;
    for(int i=0;i<2;i++)
    {
        for(int j=0;j<3;j++)
        {
            cout<<rarray[i][j]<<' ';
        }
        cout<<endl;
    }
    cout<<8<<' '<<3<<endl;
    cout<<"Your answer is?";
    cin>>answer;
    cout<<endl;
    if(answer==4)
    {
        cout<<result[0];
    }
    else
```

```

    {
        cout<<result[1];
    }
}

```

运行程序:

```

Hey!Clever boy,can you help me complete the puzzle?
Please find the rule in the numbers, and fill in the gap with an appropriate
number
6 7 2
1 5 9
8 3
Your answer is?
输入 4
Congratulations!You've done an excellent job!
输入 3
Oh,my god!You're wrong!

```

(2) 引用初始化后不能再使其成为其他变量的引用,例如:

```

int j, k;
int & i = j;
i = k; // 错误,不能更改

```

(3) 引用类似一个常量指针(int * const p),不能修改引用的指向。举例如下:

```

#include <iostream>
using namespace std;
void main()
{
    int a=32,b=25;
    int &ref=a;
    cout<<ref<<endl;
    ref = b;           //将 b 的值赋给 a 而不是将 a 的引用改为对 b 的引用
    cout<<ref<<endl;
    cout<<a<<endl;
}

```

程序运行结果如下:

```

32
25
25

```

假设有如下定义:

```

int j;
int & i = j;

```


那么, `&i` 应该是什么呢? 是一个“引用的地址”么? 答案是: `&i = &j`, 就是 `j` 这个变量的地址。

9.16 引用与指针对比

引用是 C++ 中的概念, 初学者容易把引用和指针混淆在一起。以下程序中, `n` 是 `m` 的一个引用(reference), `m` 是被引用物(referent)。

```
int m;  
int &n = m;
```

`n` 相当于 `m` 的别名(绰号), 对 `n` 的任何操作就是对 `m` 的操作。例如有人名叫王小毛, 他的绰号是“三毛”。说“三毛”怎么怎么的, 其实就是对王小毛说三道四。所以 `n` 既不是 `m` 的副本, 也不是指向 `m` 的指针, 其实 `n` 就是 `m` 它自己。

引用的一些规则如下。

- 引用被创建的同时必须被初始化(指针则可以在任何时候被初始化)。
- 不能有 NULL 引用, 引用必须与合法的存储单元关联(指针则可以是 NULL)。
- 一旦引用被初始化, 就不能改变引用的关系(指针则可以随时改变所指的对象)。

以下示例程序中, `k` 被初始化为 `i` 的引用。语句 `k=j` 并不能将 `k` 修改成为 `j` 的引用, 只是把 `k` 的值改变成 6。由于 `k` 是 `i` 的引用, 所以 `i` 的值也变成了 6。

```
int i = 5;  
int j = 6;  
int &k = i;  
k = j; // k 和 i 的值都变成了 6
```

上面的程序看起来像在玩文字游戏, 没有体现出引用的价值。

对比 9.15 节的几个例子, 会发现“引用传递”的性质像“指针传递”, 而书写方式像“值传递”。实际上“引用”可以做的任何事情“指针”也都能够做, 为什么还要“引用”这东西?

答案是: “用适当的工具做恰如其分的工作。”

指针能够毫无约束地操作内存中的任何东西, 尽管指针功能强大, 但是非常危险。就像一把刀, 它可以用来砍树、裁纸、修指甲、理发等, 谁敢这样用?

如果的确只需要借用一下某个对象的“别名”, 那么就用“引用”, 而不要用“指针”,

以免发生意外。比如说，某人需要一份证明，本来在文件上盖上公章的印子就行了，如果把取公章的钥匙交给他，那么他就获得了不该有的权利。

9.17 引用做函数的参数

引用的一个重要作用就是作为函数的参数。C/C++的函数参数是传值的，如果有大对象(例如一个大的结构)需要作为参数传递的时候，以前的(C语言中)方案往往是用指针，因为这样可以避免将整个对象全部压栈，可以提高程序的效率。但是现在(C++中)又增加了一种同样有效率的选择，就是引用。

以下是“值传递”的示例程序。由于 Func1 函数体内的 x 是外部变量 n 的一份副本，改变 x 的值不会影响 n，所以 n 的值仍然是 0。

```
void Func1(int x)
{
    x = x + 10;
}
...
int n = 0;
Func1(n);
cout << "n = " << n << endl;    // n = 0
```

以下是“指针传递”的示例程序。由于 Func2 函数体内的 x 是指向外部变量 n 的指针，改变该指针的内容将导致 n 的值改变，所以 n 的值成为 10。

```
void Func2(int *x)
{
    (* x) = (* x) + 10;
}
...
int n = 0;
Func2(&n);
cout << "n = " << n << endl;    // n = 10
```

以下是“引用传递”的示例程序。由于 Func3 函数体内的 x 是外部变量 n 的引用，x 和 n 是同一个东西，改变 x 等于改变 n，所以 n 的值成为 10。

```
void Func3(int &x)
{
    x = x + 10;
}
...
int n = 0;
```

```
Func3(n);
cout << "n = " << n << endl; // n = 10
```

与指针类型的参数一样, 引用不论指向什么类型的对象, 作为参数传递的时候都是只压栈 4 个字节(在 32 位机上)。引用所占用的 4 字节大小是根据编译器产生的代码判断的, 因为 `sizeof(a_reference)` 指令只能得到它所指向对象的大小。以下是一个相关的例子。

```
#include <iostream>
using namespace std;
void fun1(double *x)
{
    cout<<"形参的地址"<<&x<<endl;
    cout<<"占用空间大小"<<sizeof(x)<<endl; //指针占空间大小
    cout<<"变量值"<<(*x)<<endl;
}

void fun2(double &y)
{
    cout<<"形参的地址"<<&y<<endl;
    cout<<"占用空间大小"<<sizeof(y)<<endl; //变量占空间大小, 非引用
                                              //占用的空间
    cout<<"变量值"<<y<<endl;
}

void main()
{
    double a=10.0;
    cout<<"实参的地址"<<&a<<endl;
    fun1(&a);
    fun2(a);
}
```

程序运行结果如下:

```
实参的地址 0012FF78
形参的地址 0012FF28
占用空间大小 4
变量值 10
形参的地址 0012FF78
占用空间大小 8
变量值 10
```

在本例中看到的指针占用的空间大小 4 字节(在 32 机器上)是真实的指针变量作为形参占用的临时存储空间, 而 `fun2` 函数中输出的 8 字节实际上是变量 `a` 占用的空间大小, 这通过变量和引用的地址完全相同也可以判断出来。

用引用传递参数的另一个目的是为了使函数返回多个值。由于函数只能有一个返回值，若想在返回的时候传回多个值，一种方法是将多个要返回的值定义成一个结构体或类作为一个变量返回；另一种方式是将要接收返回值的变量作为函数参数以地址传递的方式传递给函数，在函数返回前给该变量赋值为要返回的值即可。在第二种方式中，既可以使用指针，也可以使用引用，但通常来说使用引用更方便。

以下示例可计算圆的面积和周长，其中函数的形参中有引用。

```
#include <iostream>
using namespace std;

bool cal(float r,float &s,float &l)
{
    s=3.14 * r * r;
    l=6.28 * r;
    return true;
}

void main()
{
    float radius=4.5,area,length;
    cal(radius,area,length);
    cout<<"圆的面积为 "<<area<<endl;
    cout<<"圆的周长为 "<<length<<endl;
}
```

程序运行结果如下：

圆的面积为 63.585

圆的周长为 28.26

当函数的形参是引用时，要注意实参也要是对应类型的变量。分析如下程序产生错误的原因。

```
int func1(int x,int y)
{
    reutn x+y ;
}
void func2(int &a)
{
    a = a*2 ;
}
void main()
{
    int a = 3,b =4 ;
    func2(func1(a,b));
}
```

本程序中 func1 是个普通函数，计算两个数的和；而 func2 是将一个数乘上 2，通过引用返回结果。但是当编译程序的时候，编译器会提示“无法从“int”转换为“int &””。这是因为，func1 返回的不是一个变量，而是一个值，因此编译器不知道 func2 中的参数 a 该引用谁，所以程序出错，这样的错误，初学者要特别小心。

9.18 应用举例 3

继续分析 9.6 节中设计的栈，在前几次修改中，已经把在栈空间满的情况下的异常处理完毕，但是在栈已经空了的情况下，若试图取数据，一样会产生错误，因为当栈空的时候，pos 的值是 0，而取数据的程序是 `return data[--pos]`；使用的是前--，因此计算机实际获取的是：`data[-1]`的数据，但这个位置是不可访问的，因此程序出错。那么如何保证能在正常情况下取得数据，在出错时候又能返回错误信息供调用者判断呢？除了通过指针还有别的办法么？答案是：引用。按照如下的方式修改程序：

```
int *data = 0;
int pos = 0;
int size;
bool init(int length)
{
    size = length;
    data = new int[length];
    If(data == NULL)
    {
        return false;
    }
    else
        return true;
}
void del()
{
    if(data != 0)
    {
        delete []data;
    }
}
bool push(int a)
{
    If(pos==size)
    {
        return false;
    }
}
```



```
    }  
    data[pos++] = a;  
    return true;  
}  
bool pop(int &t)  
{  
    if(pos>0)  
    {  
        t = data[--pos];  
        return true;  
    }  
    else  
        return false;  
}
```

注意字体加粗了的 pop 函数，这个和之前的函数不同，这个函数有个引用类型的参数，这样当需要调用的时候，可以写成如下的方式：

```
int a;  
bool re = pop(a);  
if(re)  
{  
    //正常程序  
}  
else  
{  
    //栈已经空了，不能再取数据了  
}
```

通过这样的改造，就不会出现栈已经空了仍继续访问非法内存的程序，程序的安全性又有了进一步的提高。

9.19 返回引用

同参数传递类似，函数返回值也可以有三种形式：返回普通变量、返回指针和返回引用。函数返回值时，要生成一个值的副本。而用引用返回值时，不生成值的副本。

下面的程序是有关引用返回的 4 种形式。

```
#include <iostream.h>  
float temp;  
float func1(float r)  
{
```

```

    temp = r*r*3.14;
    return temp;
}
float& func2(float r)
{
    temp = r*r*3.14;
    return temp;
}
void main()
{
    float a= func1(10);    //1
    float& b= func1(10);   //2:warning
    float c= func2(10);    //3
    float& d= func2(10);   //4
    cout<<a<<endl;
    cout<<b<<endl;
    cout<<c<<endl;
    cout<<d<<endl;
}

```

程序运行结果如下：

```

314
314
314
314

```

对主函数的 4 种引用返回的形式，程序的运行结果是一样的。但是它们在内存中的活动情况是各不相同的。其中变量 **temp** 是全局数据，驻留在全局数据区 **data** 中。函数 **main()**、**func1()**或 **func2()**驻留在栈区 **stack** 中。

第一种情况：见图 9-23。

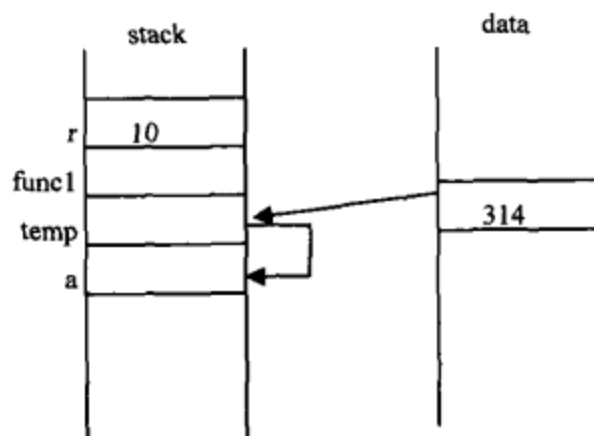


图 9-23 一般的返回值方式

这种情况是一般的函数返回值方式。返回全局变量 **temp** 值时，C++创建临时变量并将

temp 的值 314 复制给该临时变量。返回到主函数后，赋值语句 `float a=func1(10);` 把临时变量的值 314 复制给 a。

第二种情况：见图 9-24。

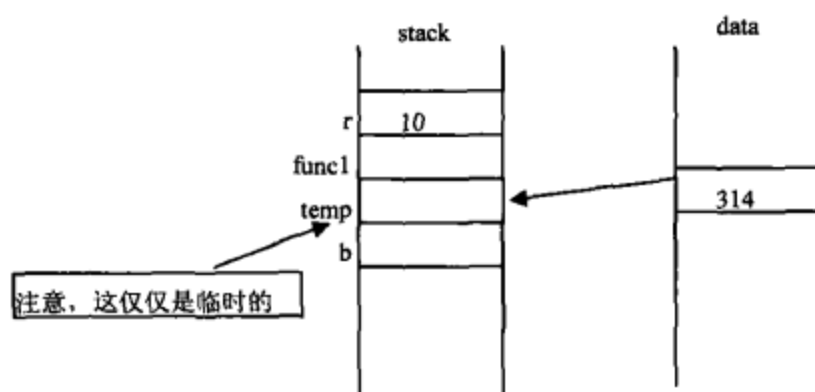


图 9-24 返回值初始引用的情形

这种情况下，函数 `func1()` 是以值方式返回的，返回时，复制 `temp` 的值给临时变量。返回到主函数后，引用 `b` 以该临时变量来初始化，使得 `b` 成为该临时变量的别名。由于临时变量的作用域短暂，所以 `b` 面临无效的危险。根据 C++ 标准，临时变量或对象的生命期在一个完整的语句表达式结束后便宣告结束，也即在 “`float& b=func1(10);`” 之后，临时变量不再存在。所以引用 `b` 以后的值是个无法确定的值。Borland C++ 对 C++ 标准进行了扩展，规定如果临时变量或对象作为引用的初始化时，则其生命期与该引用一致。这样的程序依赖于编译器的具体实现，所以移植性是差的。

若要以返回值初始化一个引用，应该先创建一个变量，将函数返回值赋给这个变量，然后再以该变量来初始化引用，就像下面这样：

```
int x=func1(5.0);
int &b=x;
```

第三种情况：见图 9-25。

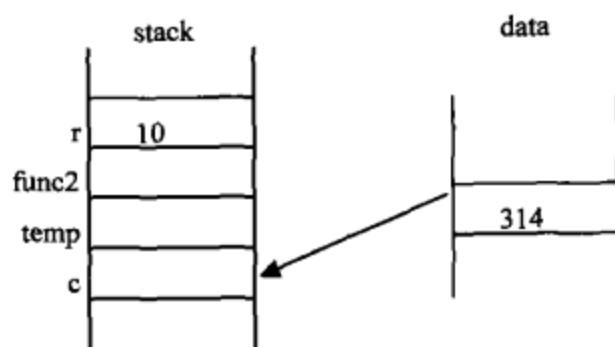


图 9-25 返回引用方式

这种情况下，函数 `func2()` 的返回值不产生副本，所以直接将变量 `temp` 返回给主函数。

主函数赋值语句中的左值，直接从变量 `temp` 中得到复制，这样避免了临时变量的产生。当变量 `temp` 是用户自定义的类型时，这种方式直接带来了程序执行效率和空间利用的提高。

第四种情况：见图 9-26。

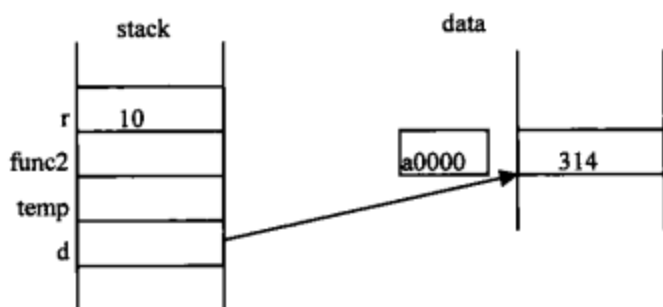


图 9-26 返回引用方式的值作为引用的初始化

这种情况下，函数 `func2()` 返回一个引用，因此不产生任何返回值的副本。在主函数中，一个引用声明 `d` 用该返回值来初始化，使得 `d` 成为 `temp` 的别名。由于 `temp` 是全局变量，所以在 `d` 的有效期内 `temp` 始终保持有效。这种做法是安全的。

但是，如果返回不在作用域范围内的变量或对象的引用，那就有问题了。这与返回一个局部作用域指针的性质一样严重。Borland C++ 中作为编译错误，Visual C++ 作为警告，来提请编程者注意。例如，下面的代码返回一个引用，来给主函数的引用声明初始化。

```
float& fn2(float r)
{
    float temp;
    temp=r*r*3.14;
    return temp;
}
void main()
{
    float &d=func2(10); //error:返回的引用是个局部变量
}
```

尽量避免返回的引用是作为一个左值进行运算的，如果程序中有下面的代码，则一定要剔除：

```
float& fn2(float r)
{
    float temp;
    temp=r*r*3.14;
    return temp;
}
void main()
{
```

```
    func2(10)=12.4; //error:返回的是局部作用域内的变量
}
```

9.20 函数调用作为左值

对于 9.19 节的第三种情况，也意味着返回一个引用使得一个函数调用表达式成为左值表达式。只要避免将局部栈中变量的地址返回，就能使函数调用表达式作为左值来使用运行得很好。

例如，下面的程序是统计学生中得奖学金和不得奖学金各占多少人。得奖学金的学生标准是总分在 300 分以上，否则不得奖学金。先看不返回引用的情况。

```
#include <iostream.h>
int array[6][4]={{60,60,60,65},
                 {75,75,75,77},
                 {80,88,90,98},
                 {80,80,54,81},
                 {62,68,69,55},
                 {95,85,97,91}};

bool getCount(int grade[], int size)
{
    int sum=0;
    for(int i=0; i<size; i++) //成绩总分
        sum+=grade[i];
    if(sum>=300)
        return true; //得奖学金
    else
        return false; //不得奖学金
}

void main()
{
    int A=0,B=0;
    int student=6;
    int gradesize=4;
    for(int i=0; i<student; i++) //处理所有的学生
        if(getCount (array[i], gradesize))
            A++;
        else
            B++;
    cout <<"得奖学金的人数是: " <<A <<endl;
    cout <<"不得奖学金的人数是: " <<B <<endl;
}
```

程序运行结果如下:

得奖学金的人数是 4

不得奖学金的人数是 2

该程序通过函数调用判明该学生成绩属于 A 类还是 B 类,然后给 A 类学生人数增量或给 B 类学生人数增量。

该程序也可以通过返回引用来实现。返回的引用作为左值直接增量。例如:

```
#include <iostream.h>

int array[6][4]={{60,80,90,75},
                 {75,85,65,77},
                 {80,88,90,98},
                 {89,100,78,81},
                 {62,68,69,75},
                 {85,85,77,91}};

int& Count(int grade[], int size,int& tA, int& tB)
{
    int sum=0;
    for(int i=0; i<size; i++) //成绩总分
        sum+=grade[i];
    if(sum>=300)
        return tA; //type A student
    else
        return tB; //type B student
}

void main()
{
    int A=0, B=0;
    int student=6;
    int gradesize=4;
    for(int i=0; i<student; i++) //处理所有的学生
        Count(array[i],gradesize,A,B)++; //函数调用作为左值
    cout <<"得奖学金的人数是:" <<A <<endl;
    cout <<"不得奖学金的人数是:" <<B <<endl;
}
```

该程序中的 Count()函数返回一个引用,为了返回一个非局部变量的引用,就要传递两个引用参数 A 和 B。当该学生属于得奖学金的类时,就返回 A 的引用,否则就返回 B 的引用。

由于返回的是引用,所以可以作为左值直接进行增量操作。该函数调用代表 A 还是 B 的左值视具体的学生成绩统计结果而定。

本例说明：返回引用的函数，可以使函数成为左值。在后面章节中会看到这一应用是很多的，典型的有前++操作符重载。

上面各种应用形式，对引用的表示依赖于实现。引用变量概念上是不占空间的，引用变量被理解为粘附在初始化的实体上，它的实现对用户来说不可见。但这并不等于具体实现的时候，非得不占任何空间。为了帮助理解引用，让读者有个引用实现的感性认识，所以用一种实现的方案，其中引用空间用来存放所代表变量的地址。

9.21 const 限定的引用

为了提高效率，可以将函数声明改为 `void Func(A &a)`，因为“引用传递”仅借用了参数的别名而已，不需要产生临时对象，但与此同时却增加了修改参数中数据的风险。函数 `void Func(A &a)` 存在一个缺点：“引用传递”有可能改变参数 `a`，这不是用户期望的。下面是一个相关的例子。

```
#include <iostream>
using namespace std;
long Func(int & data)
{
    //data--;
    long s=data;
    data--;
    if(data<1)
        return 1;
    else
        return s*Func((data));
}

int main()
{
    int a = 5;
    long result;
    result = Func(a);
    cout<<a<<endl;
    cout<<result<<endl;
    return 0;
}
```

在上面的例子中，由于错误的递减操作使函数 `Func(int)` 结束时，变量 `a` 的值已经发生了变化。解决这个问题很容易，在引用前加 `const` 修饰即可，因此函数最终成为 `void Func(const A &a)`。

当试图对常引用代表的变量进行修改时编译器会报错。相关的例子如下：

```
#include <iostream>
using namespace std;

long Func(const int & data)
{
    //data--;
    long s=data;
    data--;
    if(data<1)
        return 1;
    else
        return s*Func((data));
}

int main()
{
    int a = 5;
    long result;
    result = Func(a);
    cout<<a<<endl;
    cout<<result<<endl;
    return 0;
}
```

运行时系统报错，显示信息为：

error C2166: 1 值指定常数对象

正确的程序应为：

```
#include <iostream>
using namespace std;
long Func(const int & data)
{
    //data--;
    int s=data-1;
    data--;
    if(data<1)
        return 1;
    else
        return s*Func((data));
}
int main()
{
    int a = 5;
```



```
    long result;
    result = Func(a);
    cout<<a<<endl;
    cout<<result<<endl;
    return 0;
}
```

9.22 返回堆中变量的引用

对引用的初始化，可以是变量，可以是常量，也可以是一定类型的堆空间变量。但是，由于引用不是指针，所以下面的代码用直接从堆中获得的变量空间来初始化引用是错的：

```
int& a=new int(2); //a 不是指针
```

考虑操作符 `new`。如果 `new` 操作符不能在堆空间成功地获得内存分配，则它返回 `NULL`。因为引用不能是 `NULL`，在程序确认它不是 `NULL` 之前，程序不能用这一内存初始化引用。

例如，下面的代码说明如何处理这一校验。

```
#include<iostream.h>
void fn()
{
    int * pInt=new int;
    if(pInt==NULL)
    {
        cout<<"error memory allocation!";
        return;
    }
    int& rInt*pInt;
}
```

`int` 的指针 `pInt` 获得 `new` 返回的值，程序测试 `pInt` 中的地址，如果它是 `NULL`，则报告错误信息并返回；如果它不是 `NULL`，则将 `*pInt` 初始化引用 `rInt`。如此，`rInt` 便成为了 `new` 操作符返回的别名。

若用堆空间来初始化引用，则要求该引用在适当时候能释放堆空间。

下面的程序在堆中分配空间、求值。然后释放堆空间。

```
#include <iostream.h>
int CircleArea()
{
    double* pd=new double;
    if(!pd){
```

```
    cout <<"error memory allocation!";
    return true;
double& rd=*pd;
cout <<"the radius is: ";
cin >>rd;
cout<<"the area of circle is "<<rd*rd*3.14 <<endl;
delete &rd;
return false;
}
void main()
{
    if(CircleArea())
        cout <<"program failed.\n";
    else
        cout <<"program succeeded.\n";
}
```

程序运行结果如下:

```
the radius is: 12
the area of circle is 452.16
program succeeded.
```

在计算圆面积的 CircleArea()函数中, double 指针接受 new 返回的堆空间地址, 然后进行有效性校验。如果有效, 则将*pd 初始化引用 rd, rd 接受键盘输入, 计算, 打印输出圆面积, 返还堆空间, 并正常返回; 否则输出错误信息, 返回出错标志。

比例中返还堆空间有两种方式, 一个是 delete pd, 另一个是 delete &rd, 因为&rd 和 pd 都指向同一个堆空间地址。对引用来说, 同样存在由一个函数建立的堆内存由另一个函数释放的问题。

对使用堆的引用, 有下面的经验。

- 必要时用值传递参数。
- 必要时返回值。
- 不要返回有可能退出作用域的引用。
- 不要引用空目标。

引用和指针使函数的“黑盒”性被打破。函数可以访问不属于自己栈空间的内存。这对把握不住 C++的人来说是危险的, 而对熟练的程序员来说, 正是引用和指针才使函数只能返回单一值的状态被打破, 使得函数功能更趋强大。函数的副作用是良性还是恶性, 各自有评说, 对于函数潜在的破坏性, 是放任自流, 还是要适当地抑制, 专家们动尽了脑筋,

直到 C++ 类机制的实现, 才使函数的恶性作用得以控制。

本章小结

本章讲述了 C 类语言的灵魂: 指针和引用, 其中内存的分配与释放是重点, 而函数与指针的结合引用, 是程序设计中的常用方式。

习 题

1. 定义一个指向字符串的指针数组, 用一个函数完成 N 个不等长字符串的输入, 根据实际输入的字符串长度分配存储空间, 依次使指针数组中的元素指向每一个输入的字符串。设计一个完成 N 个字符串按升序排序的函数(在排序过程中, 要求只交换指向字符串的指针值, 不交换字符串)。在主函数中实现将排序后的字符串输出。

2. 编写求一维数组平均值的函数, 用引用类型变量作为函数参数返回平均值。在主函数中输入数组元素值, 并求出平均值。

3. 输入一个二维数组 `a[6][6]`, 设计一个函数, 用指向一维数组的指针变量和二维数组的行数作为函数的参数, 求出平均值、最大值和最小值, 并输出。

4. 请问下面两个例子的运行结果一样吗?

```
void func(int *pInt)
{
    int k=4;
    pInt=&k;
}
void main()
{
    int k=3;
    int *ptr=&k;
    func(ptr);
    cout<<(*ptr)<<endl;
    system("pause");
}

void func(int *pInt)
{
    (*pInt)=4;
```




```
}  
void main()  
{  
    int k=3;  
    int *ptr=&k;  
    func(ptr);  
    cout<<(*ptr)<<endl;  
    system("pause");  
}
```

程序员 知识 库
PDG

第 10 章 结构、联合、枚举

本章内容:

- 自定义数据类型的必要性。
- 结构体的应用。
- 联合和枚举类型。
- 链表的应用。

重点:

- 链表的创建。
- 链表的应用。

目的:

通过自定义数据类型, 优化数据结构, 调整程序结构; 通过链表的应用, 接触线性表这样的数据结构的定义及使用。

10.1 自定义数据类型概述

1. 为什么要使用自定义数据类型

虽然 C++ 提供了 int、long、double 和 float 等多种基本数据类型, 但是这还是远远不能满足实际工作的要求。因此 C++ 提供了强大的自定义数据类型功能, 用户可以根据需要创建恰当的数据类型。

例如, 一个小组由 5 名学生组成, 为了记录学生的成绩及对成绩进行排序, 需要记录每个人的学号、姓名、年龄和成绩, 对于这些数据的存储可以用多个一维数组进行处理, 程序如下:

```
#include <iostream>
using namespace std;
void main()
{
```

```

char name[5][20] = {"张三", "李四", "王五", "令狐冲", "风清扬"};
int num[5] = {1, 2, 3, 4, 5};
int age[5] = {22, 22, 23, 25, 23};
float score[5] = {81, 82.5, 79, 90, 91.5};
int i;
for(i = 0; i < 5; i++)
{
    cout << "姓名: " << name[i] << "学号: " << num[i] << "年龄: " << age[i] << "成绩: " << score[i] << endl;
}
}

```

程序运行结果如下:

```

姓名: 张三 学号: 1 年龄: 22 成绩: 81
姓名: 李四 学号: 2 年龄: 22 成绩: 82.5
姓名: 王五 学号: 3 年龄: 23 成绩: 79
姓名: 令狐冲 学号: 4 年龄: 25 成绩: 90
姓名: 风清扬 学号: 5 年龄: 23 成绩: 91.5

```

这样的程序在操作数据时,就要时刻记住数组每一维的每个分量代表什么。当有很多类似于这种需要多个分量共同存储的时候,记忆每个分量的含义将是一件非常繁琐且易出错的事情。此时就需要用到自定义数据类型的变量。

2. 自定义数据类型的分类

C++中的自定义数据类型可以分为四类:结构、联合、枚举和类。以下几节将详细讨论前三种的使用,对于类的讲解放到第11章及其以后部分。

10.2 结构的定义

结构与数组类似,都是由若干分量组成的。数组是由相同类型的数组元素组成,但结构的分量可以是不同类型的,结构中的分量称为结构的成员。访问数组中的分量(元素)是通过数组的下标,而访问结构中的成员是通过成员的名字。

在程序中使用结构之前,首先要对结构的组成进行描述,即在使用结构变量时要先定义结构类型。定义结构类型常见以下两种格式。

格式1:

```

struct 结构名
{
    变量类型 变量1;

```

```
    变量类型 变量 2;  
    ...  
};
```

格式 2:

```
typedef struct  
{  
    变量类型 变量 1;  
    变量类型 变量 2;  
    ...  
}结构名;
```

结构名是结构类型的类型标识符而不是变量名, 结构名的构成规则与变量名相同。在结构体内部的变量, 称作成员变量。用已说明的结构类型就可定义结构变量了。定义格式与普通变量相同:

结构名 变量名;

成员类型可以是五种基本数据类型(整型、浮点型、字符型、指针型和 void 型)的变量、数组, 或者是另一种自定义类型变量(比如结构), 一个结构的成员可以是另一个结构类型的变量, 在后面将会详细分析这种情况。下面来看看怎样定义结构变量。第一种方法是先定义结构类型, 再定义结构变量, 示例如下。

```
struct person  
{  
    char name[8];  
    int age;  
    char sex[2];  
    char depart[20];  
    float wage1, wage2;  
};  
person somebody;①
```

第二种方法是在定义结构类型的同时定义结构变量, 格式如下:

```
struct 结构名  
{  
    成员类型 成员变量 1;  
    成员类型 成员变量 2;  
    ...  
};
```

① 本书讲述的是 C++, 但是结构是在 C 中就有的, 部分读者可能学习过 C 语言, 因此要注意如果是 C 编译器, 若用格式 1 声明结构, 则此语句是错的, 必须按照格式 2 声明结构, 但在 C++ 中, 该语句是正确的。

} 结构变量;

这时上面的例子可改为以下形式。

```
struct string
{
    char name[8];
    int age;
    char sex[2];
    char depart[20];
    float wage1, wage2;
} person;
```

这个例子定义了一个结构名为 `string` 的结构变量 `person`，如果省略变量名 `person`，则变成对结构的说明。

如果需要定义多个具有相同形式的结构变量时用这种方法比较方便，它先做结构说明，再用结构名来定义变量。

例如：

```
string Tianyr, Liuqi, ...;
```

如果省略结构名，则称之为无名结构，这种情况常常出现在函数内部，用这种结构时前面的例子变成：

```
struct
{
    char name[8];
    int age;
    char sex[2];
    char depart[20];
    float wage1, wage2;
} a, b;
```

10.3 结构初始化

由于结构体类型变量汇集了各类不同数据类型的成员，所以结构体类型变量的初始化就略显复杂。

下面通过一个例子来介绍结构体类型变量的定义和初始化。

```
struct student
{
    char name[20];
```

```

    int num;
    int age;
    float score;
};
student stu = {"张三", 1, 22, 81};

```

结构体类型变量完成初始化后，其各成员的值分别为：

```

stu.name = "张三"
stu.num = 1
stu.age = 22
stu.score = 81

```

其存储在内存的情况如图 10-1 所示。

张三	1	22	81
----	---	----	----

图 10-1 结构体类型变量在内存中的存储

可以通过 C++ 提供的输入输出函数完成对结构体类型变量成员的输入输出。由于结构体类型变量成员的数据类型通常是不一样的，所以要将结构体类型变量成员以字符串的形式输入，要先利用 C++ 的类型转换函数将其转换为所需类型。类型转换的函数如下：

- `int atoi(char *str);`：转换 `str` 所指向的字符串为整型，其函数的返回值为整型。
- `double atof(char *str);`：转换 `str` 所指向的字符串为双精度。
- `long atol(char *str);`：转换 `str` 所指向的字符串为长整型。

使用上述函数时，要包含头文件“`stdlib.h`”。

对上述的结构体类型变量成员输入采用的一般形式的示例如下：

```

void main()
{
    char temp[20];
    gets(stu.name);
    gets(temp);
    stu.num = atoi(temp);
    gets(temp);
    stu.age = atoi(temp);
    gets(temp);
    stu.score = atof(temp);
}

```

对该结构体类型变量成员的输出也必须将各成员独立输出，而不能将结构体类型变量以整体的形式输入输出。

结构体类型同样可以定义数组，定义方式如下：

结构体类型 变量[数组大小];

数组的初始化方式与普通数组初始化一样,只是需要注意,每个数组元素有多个元素:

结构体类型 变量[数组大小]={ {数据 1,数据 2...}, {数据 1,数据 2...}...}

C++允许针对具体问题定义各种各样的结构体类型,甚至是嵌套的结构体类型。举例如下。

10.4 访问结构成员

学习了如何定义结构体类型和结构体类型变量,怎样正确地引用该结构体类型变量的成员呢? C++规定引用的形式为:

<结构体类型变量名> . <成员名>

若结构体类型及变量的定义如下:

```
struct data
{
    int day;
    int month;
    int year;
} time1,time2;
```

则变量 time1 和 time2 各成员的引用形式为:

```
time1.day
time1.month
time1.year
time2.day
time2.month
time2.year
```

变量中成员的引用形式如图 10-2 所示。

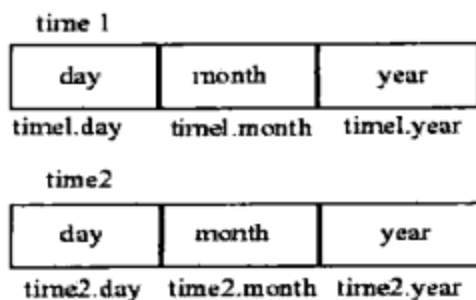


图 10-2 结构体类型示例中变量各成员的引用形式

其结构体类型变量的各成员与相应的简单类型变量使用方法完全相同。

time1 和 time2 都有各自的成员 day、month 和 year, 但要注意, time1 的成员与 time2 的成员类型是一样的, 但不是一个成员, 如同每个人都有名字, 但甲的名字和乙的名字是两码事。

```
struct date
{
    int day;
    int month;
    int year;
};
struct stu
{
    char name[20];
    date birthday;
} person;
```

该结构体类型变量成员的引用形式:

```
person.name
person.birthday.day
person.birthday.month
person.birthday.year
```

10.5 结构与数组

结构与数组的关系有两重: 其一是在结构中使用数组类型作为结构的一个成员; 其二是用结构类型作为数组元素的基类型构成数组。前者在前面的例题中已多次见到; 后者是本节要讨论的内容。

结构数组是一个数组, 其数组中的每一个基本元素都是结构类型。说明结构数组的方法是: 先定义一个结构, 然后用结构类型说明一个数组变量。

例如: 为记录 100 个人的基本情况, 可以说明一个有 100 个元素的数组, 每个元素的基类型为一个结构, 在说明数组时可以写成:

```
student stu[100];
```

stu 就是有 100 个元素的结构数组, 数组的每个元素为 student 型结构。

要访问结构数组中的具体结构, 必须遵守数组使用的规定, 按数组名及其下标进行访问; 要访问结构数组中某个具体结构下的成员, 又要遵守有关访问结构成员的规定, 使用

“.” 访问运算符和成员名。访问结构数组成员的一般格式是：

结构数组名[下标].成员名

同一般的数组一样，结构数组中每个元素的起始下标从 0 开始，数组名称表示该结构数组的存储首地址。结构数组存放在一连续的内存区域中，它所占内存数目为结构类型的大小乘以数组元素的个数。结构数组 `man` 在内存中的存储如图 10-3 所示。

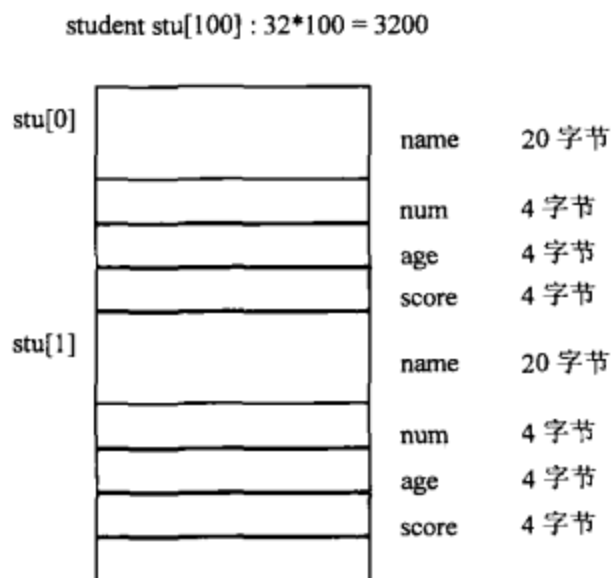


图 10-3 结构数组 `man` 在内存中的存储

例如，要将数组 `man` 中的 1 号元素赋值为：“test”,9,19,83，就可以使用下列语句。

```
strcpy(stu[1].name, " test");
stu[1].num = 9;
stu[1].age = 19;
stu[1].score = 83; //为结构数组中一个元素的各个成员赋值
```

现在改用结构再来描述 10.1 节中学生小组的数据信息，其程序如下。

```
#include <iostream>
using namespace std;
struct student
{
    char name[20];
    int num;
    int age;
    float score;
};
void main()
{
    student stu[5] = {{"张三",1,22,81},
                      {"李四",2,22,82.5},
                      {"王五",3,23,79}}
```

```

        {"令狐冲",4,25,90}
        {"风清扬",5,23,91.5}};

int i;
for(i = 0; i<5; i++)
{
    cout<<"姓名: "<< stu[i].name<<"学号: "<<stu[i].num<<"年龄: "
        <<stu[i].age<<"成绩: "<<stu[i].score<<endl;
}
}

```

程序运行结果如下:

```

姓名: 张三 学号: 1 年龄: 22 成绩: 81
姓名: 李四 学号: 2 年龄: 22 成绩: 82.5
姓名: 王五 学号: 3 年龄: 23 成绩: 79
姓名: 令狐冲 学号: 4 年龄: 25 成绩: 90
姓名: 风清扬 学号: 5 年龄: 23 成绩: 91.5

```

在本例中,使用到了自定义数据类型: `student`。这样写程序对每个人的数据可看作一个独立单位程序会很清晰。

以下示例将设计一个简单的密码加密程序。

分析:加密程序的加密过程是先定义一张字母加密对照表,将需要加密的一行文字输入加密程序,程序根据加密表中的对应关系,可以很简单地将输入的文字加密输出,对于表中未出现的字符则不加密。

输入字符与输出字符的对应关系如图 10-4 所示。

输入	输出	输入	输出
c	k	a	b
b	w	d	:
e	i	i	a
k	b	:	c
w	e		

图 10-4 输入字符与输出字符的对应关系

可以定义一个结构来表示加密表。结构 `table` 中的成员 `input` 存入输入的字符,成员 `output` 保存加密后对应的字符,其程序如下。

```

#include "stdio.h"
struct table
{
    char input;
    char output;
}
//定义结构 table
//成员 input 存输入的字符
//成员 output 存输出的字符

```

```

};
struct table translate[ ]=          //说明外部的结构数组 translate 并初始化
{ 'a', 'd', 'b', 'w', 'c', 'k', 'd', ';', 'e', 'i', 'i', 'a', 'k', 'b', ';',
  'c', 'w', 'e' };
//建立加密对照表
main( )
{
    char ch;
    int str_long, i;
    str_long = sizeof(translate)/sizeof(struct table); //计算数组元素个数
    while ( (ch=getchar( )) != '\n')
    {
        for ( i=0; translate[i].input!=ch && i<str_long; i++) ;
        {
            if (i<str_long)
                putchar(translate[i].output);    //对表中的字符加密输出
            else
                putchar (ch);                    //对其他字符原样输出
        }
    }
}

```

语句“`struct table translate[]={...}`”有三个作用，一是说明了一个外部的结构数组 `translate`；二是表示数组的大小由后面给出的初始化数据决定；三是对结构数组进行初始化。在程序中给出了数组初始化数据，所以结构数组 `translate` 有 9 个元素。

程序中语句“`str_long = sizeof(translate)/sizeof(struct table);`”是用 `sizeof` 运算计算结构数组 `translate` 中元素的数目。`sizeof(translate)` 求出数组 `translate` 所占用的字节总数，`sizeof(struct table)` 求出数组中每个元素所占用的字节总数。

运行程序时，从键盘上逐个读取输入的字符存入变量 `ch` 中，将 `ch` 的值与结构 `translate` 中的 `input` 比较，如果是要加密的字符，则输入加密后的字符 `output`，否则 `ch` 原样输出。

下面这个例子用于求解约瑟夫问题，其内容是有 n 个小孩围成一圈并依次编号，教师指定从第 m 个小孩开始报数，当报到第 s 个小孩时，即令其出列，然后再从下一个孩子起从 1 开始继续报数，数到第 s 个小孩又令其出列，这样直到所有的孩子都依次出列。求小孩出列的顺序。分析：由于问题中的小孩围成一个圈，因而可以用一个环形链来表示。用结构数组构成一个环形链，结构名为 `child`，数组名为 `link`。`nextp` 的含义是排在当前这个孩子后面的下一个孩子的序号。由 `nextp` 可构成一个环型链，`no` 是孩子的序号。这样就可以从第 m 个小孩开始沿着 `nextp` 连成的闭合链不断计数 s 次，输出对应的 `no` 表示让他出列，程序如下。

```

struct child          //定义结构 child

```

```

{
    int nextp;                //排在后面下一个位置上的孩子的序号
    int no;                   //孩子的序号
} link [100];                //说明结构数组 link
main( )
{
    int i, n, s, y, k, m, count=0; //count 为输出计数器
    cout<<endl<<"Tell me how many children are there ?";
    cin>>n;
    cout<<endl<<"From which to count ?";
    cin>>m;
    cout<<endl<<"How many shall I count ?";
    cin>>s;
    for ( i=1; i<=n; i++ )    //根据孩子总数 n 建立一个环
    {
        if ( i==n )
            link[i].nextp=1;    //若是最后一个, 则他的下一个是第一个人
        else
            link[i].nextp=i+1;    //否则, 第 i 个人的下一个为第 i+1 个人
        link[i].no=i;            //为第 i 个孩子建立序号
    }
    cout<<endl<<"Stand out : " <<endl;
    if ( m>=i )
        k=n;
    else
        k=m-1;                //k 定位在应开始计数的孩子的前一个人上
    while (count != n)
    {
        for (i=0; i!=s; )    //s 个孩子报数
        {
            k = link[k].nextp;    //取下一个孩子
            if ( link[k].no != 0 )
                ++i;                //若序号不为 0 表示没出列, 则计数
        }
        cout<<link[k].no;        //输出出列的孩子序号
        link[k].no = 0;          //将出列孩子的序号清为 0
        if ( ++count % 10 == 0 )
            cout<<endl;
    }
}

```

程序中首先输入小孩总数 n 、报数起始位置 m 和需要出列的计数步长 s , 然后由总数 n 初始化结构数组 `link`。为了程序中处理的方便, `link` 数组从下标 1 开始使用, 下标为 0 的元素不用, 由于数组说明长度为 100, 所以输入的 n 值不能大于 99。

变量 `k` 记录下一次需要计数的数组下标；初始化时，变量 `k` 的值在要开始报数的小孩的前一个位置。在 `while` 循环中，使用 `count` 作为出列小孩计数器，对于已出列的小孩，将 `no` 清为 0；变量 `i` 是报数计数器，当 `link[k].no` 不为 0 时，才操作 `++i`。语句“`k=link[k].nextp;`”的含义是将下一个孩子所在数组中的下标位置送入变量 `k` 中。

运行程序，输入 `n=35、m=5、s=3`，可得到如下结果：

```
stand out
7 10 13 16 19 22 25 28 31 34
2 5 9 14 18 23 27 32 1 6
12 20 26 33 4 15 24 35 11 29
8 30 21 3 17
```

本例完全可以用一维数组实现，请读者自己编写程序。

下面是结构体数组的应用举例。对候选人得票的统计程序，设有三个候选人，每次输入一个候选人的名字，最后统计出每个候选人的得票结果。

分析：先定义结构体数组，其元素是结构体，含两个成员，一个是候选人的姓名，另一个是候选人的得票数。

```
struct person
{
    char name[20];int count;
}leader[3]={"Li",0,"Zhang",0,"Fun",0};

main()
{
    int i,j;
    char leader_name[20];
    for(i=1;i<=10;i++)
    {
        cin>>leader_name;        //输入候选人的名字
        for(j=0;j<3;j++)
            if(strcmp(leader_name,leader[j].name)==0)
                leader[j].count++; //统计候选人的得票数
    }
    cout<<endl;
    for(i=0;i<3;i++)
        cout<<leader[i].name<<" "<<leader[i].count;
}
```

10.6 结构与指针

当一个指针变量用来指向一个结构变量时，称之为结构指针变量。

结构指针变量中的值是所指向的结构变量的首地址。通过结构指针即可访问该结构变量，这与数组指针和函数指针的情况是相同的。结构指针变量说明的一般形式为：

```
struct 结构名 *结构指针变量名
```

例如，若定义了 student 这个结构，要说明一个指向 student 的指针变量 pstu，可写为：

```
student *pstu;
```

当然也可在定义 student 结构时同时说明 pstu。与前面讨论的各类指针变量相同，结构指针变量也必须要先赋值后才能使用。赋值是把结构变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。如果 boy 是被说明为 student 类型的结构变量，则：pstu=&boy 是正确的，而：pstu=&stu 是错误的。

结构名和结构变量是两个不同的概念，不能混淆。结构名只能表示一个结构形式，编译系统并不对它分配内存空间。只有当某变量被说明为这种类型的结构时，才对该变量分配存储空间。因此上面 &stu 这种写法是错误的，不可能去取一个结构名的首地址。有了结构指针变量，就能更方便地访问结构变量的各个成员，其访问的一般形式为：

(*结构指针变量).成员名

或为：

结构指针变量->成员名

例如：

```
(*pstu).num
```

或者：

```
pstu->num
```

应该注意(*pstu)两侧的括号不可少，因为成员符“.”的优先级高于“*”。如去掉括号写作*pstu.num 则等效于*(pstu.num)，这样意义就完全不对了。下面通过例子来说明结构指针变量的具体说明和使用方法。

```
struct student
{
    char *name;
    int num;
```

```

int age;
float score;
} boy1={"张三",10,23, 78.5},*pstu;
main()
{
    pstu=&boy1;
    cout<<pstu->name<<pstu->num<< pstu->age<< pstu->score<<endl;
    cout<<*(pstu).name<<*(pstu). num<< *(pstu).age<< *(pstu).score<<endl;
    cout<< boy1.name<< boy1. num<< boy1.age<< boy1.score<<endl;
}

```

本例程序定义了一个结构 `student`，定义了 `student` 类型结构变量 `boy1` 并作了初始化赋值，还定义了一个指向 `student` 类型结构的指针变量 `pstu`。在 `main` 函数中，`pstu` 被赋予 `boy1` 的地址，因此 `pstu` 指向 `boy1`。然后在 `cout` 语句内用三种形式输出 `boy1` 的各个成员值。从运行结果可以看出：

结构变量.成员名
 (*结构指针变量).成员名
 结构指针变量->成员名

这三种用于表示结构成员的形式是完全等效的。结构数组指针变量可以指向一个结构数组，这时结构指针变量的值是整个结构数组的首地址；结构指针变量也可指向结构数组的一个元素，这时结构指针变量的值是该结构数组元素的首地址。设 `ps` 为指向结构数组的指针变量，则 `ps` 也指向该结构数组的 0 号元素，`ps+1` 指向 1 号元素，`ps+i` 则指向 `i` 号元素。这与普通数组的情况是一致的。

以下示例用指针变量输出结构数组。

```

#include <iostream>
using namespace std;
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}boy[5]={
    {101,"Zhou ping",'M',45},
    {102,"Zhang ping",'M',62.5},
    {103,"Liou fang",'F',92.5},
    {104,"Cheng ling",'F',87},
    {105,"Wang ming",'M',58},
};
void main()
{
    struct stu *ps;

```



```

        cout<<"No\tName\t\t\tSex\tScore\t\n";
        for (ps=boy;ps<boy+5;ps++)

        cout<<ps->num<<'\\t'<<ps->name<<"\\t\\t"<<ps->sex<<'\\t'<<ps->score<<'\\t'<<endl;
    }

```

在程序中，定义了 stu 结构类型的外部数组 boy 并作了初始化赋值。在 main 函数内定义 ps 为指向 stu 类型的指针。在循环语句 for 的表达式 1 中，ps 被赋予 boy 的首地址，然后循环 5 次，输出 boy 数组中各成员值。应该注意的是，一个结构指针变量虽然可以用来访问结构变量或结构数组元素的成员，但是，不能使它指向一个成员。也就是说不允许取一个成员的地址来赋予它。因此，下面的赋值是错误的：“ps=&boy[1].sex;”而只能是：ps=boy;(赋予数组首地址)或者是：ps=&boy[0];(赋予 0 号元素首地址)。

10.7 结构与引用

结构是自定义数据类型，一种结构一经定义，便可以同普通的数据类型同样使用。结构体变量与普通变量一样，可以声明对结构变量的引用。声明的格式为：

结构名 &引用名=结构变量；

例如：

```

student zhangsan;
student &monitor=zhangsan;

```

利用引用访问结构体变量成员的访问方式和直接使用结构变量是相同的。

对于 10.5 节中用结构描述学生小组的数据信息的例子改用引用来访问变量，其代码如下：

```

#include <iostream>
using namespace std;
struct student
{
    char name[20];
    int  num;
    int  age;
    float score;
};
void main()
{
    student stu[5] = {{"张三",1,22,81},
                      {"李四",2,22,82.5}
                      {"王五",3,23,79}

```



```

        {"令狐冲",4,25,90}
        {"风清扬",5,23,91.5}}};
student (&ref)[5]= stu;
cout<<"姓名\t学号\t\t\t\t年龄\t成绩\t"<<endl;
for(int i=0;i<5;i++)
{
    cout<<ref[i].name<<' '\t'<<ref[i].num<<"\t\t"<<ref[i].age<<' '\t'
    <<ref[i].score <<endl;
}
}

```

程序定义了对结构数组的引用,然后利用引用对结构数组的元素依次进行了输出。程序运行结果为:

姓名	学号	年龄	成绩
张三	1	22	81
李四	2	22	82.5
王五	3	23	79
令狐冲	4	25	90
风清扬	5	23	91.5

10.8 在函数中使用结构

与其他数据类型一样,也可以将结构作为参数传递给函数。下面的程序使用一个函数将传递过来的结构中的信息显示到屏幕上。

```

#include <iostream>
#include <iomanip>
using namespace std;
struct data
{
    float amount;
    char fname[30];
    char lname[30];
} rec;
void print_rec(struct data x);

int main(void)
{
    cout<<"Enter the donor's first and last names"<<endl;
    cout<<"separated by a space: ";
    cin>>rec.fname>>rec.lname;
    cout<<endl<<"Enter the donation amount: ";
    cin>>rec.amount;
}

```

```
    //调用 display 函数
    print_rec( rec );
    return 0;
}

void print_rec(struct data x)
{
    cout<<endl<<"Donor "<<x.fname<<' ' <<x.lname<<" gave $";
    cout<<setiosflags(ios:: fixed);
    cout<<setprecision(2)<<x.amount<<endl;
}
```

程序运行结果如下:

Enter the donor's first and last names
separated by a space: zhan xp

Enter the donation amount: 34.5

Donor zhan xp gave \$34.50

使用结构作为函数参数的最大好处是解决了传入参数过多的问题。当向函数传递参数时,参数个数过多会增加程序员记忆参数类型和目的的负担,而把多个参数归结为一个结构,在调用函数前填充这样一个结构变量不仅简化了函数的定义,而且由于结构的成员可以根据名称访问,这大大降低了程序员的记忆负担。通常,由于变量占用空间较大,采用值传递的方式会因实参和形参间的复制而降低效率,故在使用结构传递参数时,多以结构指针或结构的引用为主。

前面的例子改成使用结构指针传参的形式,其程序代码如下:

```
#include <iostream>
#include <iomanip>
using namespace std;
struct data{
    float amount;
    char fname[30];
    char lname[30];
} rec;

void print_rec(const data *x);

int main(void)
{
    cout<<"Enter the donor's first and last names"<<endl;
    cout<<"separated by a space: ";
    cin>>rec.fname>>rec.lname;
```

```

        cout<<endl<<"Enter the donation amount: ";
        cin>>rec.amount;
        print_rec( &rec );      //用结构指针传递参数
        return 0;
    }

void print_rec(const data *x)
{
    cout<<endl<<"Donor "<<x->fname<<' ' <<x->lname<<" gave $";
    cout<<setiosflags(ios:: fixed);
    cout<<setprecision(2)<<x->amount<<endl;
}

#include <iostream>
#include <iomanip>
using namespace std;

struct data{
    float amount;
    char fname[30];
    char lname[30];
} rec;

```

改成引用传递参数的形式，其程序代码如下：

```

void print_rec(const data &x);

int main(void)
{
    cout<<"Enter the donor's first and last names"<<endl;
    cout<<"separated by a space: ";
    cin>>rec.fname>>rec.lname;
    cout<<endl<<"Enter the donation amount: ";
    cin>>rec.amount;
    print_rec( rec );      //通过引用传递参数
    return 0;
}

void print_rec(const data &x)
{
    cout<<endl<<"Donor "<<x.fname<<' ' <<x.lname<<" gave $";
    cout<<setiosflags(ios:: fixed);
    cout<<setprecision(2)<<x.amount<<endl;
}

```

另外，前面在讲函数时也提到了，利用结构作为函数的返回值可以返回多个量。

下面的程序通过函数返回一个结构值给结构变量赋值。

```
#include <iostream.h>
struct Person
{
    char name[20];
    unsigned long id;
    float salary;
};
Person GetPerson()
{
    Person temp;
    cout <<"Please enter a name for one person: \n";
    cin>>temp.name;
    cout <<"Enter one's id number and his salary: \n";
    cin >>temp.id >>temp.salary;
    return temp;
}
void Print(Person& p)
{
    cout <<p.name <<" "
         <<p.id <<" "
         <<p.salary <<endl;
}
void main()
{
    Person employee[3];
    for(int i=0; i<3; i++){
        employee[i]=GetPerson();    //通过函数返回结构值给结构变量赋值
        Print(employee[i]);
    }
}
```

程序运行结果如下:

```
Please enter a name for one person:
marit
Enter one's id number and his salary:
27519 311.0
marit 27519 311
Please enter a name for one person:
jone
Enter one's id number and his salary:
12345 339.0
jone 12345 339
Please enter a name for one person:
```

```

peter
Enter one's id number and his salary:
23987 400.0
peter 23987 400

```

主函数中调用了 `GetPerson()` 函数，该函数返回 `Person` 结构值，该结构值被赋给了主函数中的结构数组元素。

由于结构返回时，要复制结构值给一个临时结构变量(参考复制构造一节)，当结构很大时，运行效率会受影响。可以用一种效率更高的结构参数引用传递的方法来代替。结构参数引用传递时无须复制结构值，连赋值操作都不需要。

下面的程序用结构引用作函数参数的方法实现上例程序的同样功能。

```

#include <iostream.h>
struct Person
{
    char name[20];
    unsigned long id;
    float salary;
};
void GetPerson(Person& p) //结构参数引用传递的函数
{
    cout <<"Please enter a name for one person: \n";
    cin>>p.name;
    cout <<"Enter one's id number and his salary: \n";
    cin >>p.id >>p.salary;
}
void Print(Person& p)
{
    cout <<p.name <<" "
         <<p.id <<" "
         <<p.salary <<endl;
}
void main()
{
    Person employee[3];
    for(int i=0; i<3; i++){
        GetPerson(employee[i]); //调用后 employee[i] 被赋值
        Print(employee[i]);
    }
}

```

程序运行结果如下：

```
Please enter a name for One person:
```

```
marit
Enter one's id number and his salary:
27519 339.0
marit 27519 311
Please enter a name for One person:
jone
Enter One, S id number and his salary:
12345 339. 0
jone 12345 339
Please enter a name for one person:
peter
Enter one;s id number and his salary:
23987 400. 0
peter 23987 400
```

用结构参数引用传递，无须返回结构值，无须给另一个结构变量赋值，也无须在函数返回时复制结构值给临时结构变量，节省了系统开销。在主函数中，调用该函数的格式应作相应的调整。

一个函数可以返回一个结构的引用和结构的指针。但是，不要返回一个局部结构变量的引用或指针。

下面的代码不应将局部结构变量的引用返回给上层函数。

```
Person& GetPerson()
{
    Person p;
    cout<<"Please enter a name for one person: \n";
    cin.get(p.name);
    cout<<"Enter one's idnumber andhis salary: \n";
    cin>>p.id>>p.salary;
    return p; //局部结构变量的地址
}
void main()
{
    Person& sp=GetPerson();
    //...
}
```

在主函数中，引用 sp 用 GetPerson()来初始化，使得 sp 与 GetPerson()中的局部变量名同享一个空间，这是不好的程序设计。

10.9 结构的复杂形式

结构的复杂形式包括嵌套结构、位结构等。下面分别进行介绍。

1. 嵌套结构

嵌套结构是指在一个结构成员中可以包括其他一个结构，C++编译器允许这种嵌套。

下面的例子定义了一个有嵌套的结构。

```
struct addr
{
    char city[20];
    unsigned long zipcode;
    char tel[14];
}
struct student
{
    char name[8];
    int age;
    struct addr address;
} student;
```

其中：**addr** 为另一个结构的结构名，必须要先进行说明：

如果要给 **student** 结构中成员 **address** 结构中的 **zipcode** 赋值，则可写成：

```
student.address.zipcode=200001;
```

每个结构成员名从最外层直到最内层逐个被列出，即嵌套式结构成员的表达方式是：

结构变量名.嵌套结构变量名.结构成员名

其中：嵌套结构可以有很多，结构成员名为最内层结构中不是结构的成员名。

2. 位结构

位结构是一种特殊的结构，在需按位访问一个字节或字的多个位时，位结构比按位运算符更加方便。

位结构定义的一般形式为：

```
struct 位结构名{
    数据类型 变量名: 整型常数;
    数据类型 变量名: 整型常数;
```

} 位结构变量;

其中: 数据类型必须是 `int(unsigned` 或 `signed)`型。整型常数必须是非负的整数, 范围是 0~15, 表示二进制位的个数, 即表示有多少位。

变量名是选择项, 可以不命名, 这样规定是为了排列需要。

以下示例定义了一个位结构。

```
struct test
{
    unsigned a: 8;      //a 占用低字节的 0~7 位共 8 位
    unsigned b: 4;      //b 占用高字节的 0~3 位共 4 位
    unsigned c: 3;      //c 占用高字节的 4~6 位共 3 位
    unsigned d: 1;      //d 占用高字节的第 7 位
};
```

位结构成员的访问与结构成员的访问相同。

访问上例位结构中的 `c` 成员可写成:

```
test ch;
ch.c
```

注意:

- 位结构中的成员可以定义为 `unsigned`, 也可定义为 `signed`, 但当成员长度为 1 时, 会被认为是 `unsigned` 类型。因为单个位不可能具有符号。
- 位结构中的成员不能使用数组和指针, 但位结构变量可以是数组和指针, 如果是指针, 其成员访问方式与结构指针相同。
- 位结构总长度(位数), 是各个位成员定义的位数之和, 可以超过两个字节。
- 位结构成员可以与其他结构成员一起使用。

以下示例定义一个员工信息的结构。

```
struct info
{
    char name[8];
    int age;
    struct addr address;
    float pay;
    unsigned state: 1;
    unsigned pay: 1;
}workers;
```

此例的结构中有两个位结构成员, 每个位结构成员只有一位, 因此只占一个字节但保

存了两个信息。由此可见使用位结构可以节省存储空间。

10.10 链 表

数组作为存放同类数据的集合，在程序设计时有很多方便，增加了灵活性。但数组也同样存在一些弊病。如数组的大小在定义时要事先规定，不能在程序中进行调整。这样一来，在程序设计中针对不同问题有时需要 30 个大小的数组，有时需要 50 个大小的数组，难于统一，只能够根据可能的最大需求来定义数组，常常会造成一定存储空间的浪费。另外，当向结构数组插入元素和删除元素时，必须对数组中的元素作移动，这很不方便。

那么，可否这样来设计程序呢？每次需要存储数据时分配内存，然后将内存与其他已经分配的内存一起记录下来。为了解决这个问题，回想一下测量弹簧的弹力系数的实验，在实验中一个重要的实验器材是钩码，钩码的特点是每个钩码自身有一定的重量：一个圆环及一个钩，每个钩码都可以被别的钩码钩住，同时又可以钩住其他钩码。这样，只要钩住第一个钩码，即可提起所有的钩码，测试重力对弹簧的作用效果。如果每个数据也能像钩码一样可以钩住其他数据，同时也可以被其他数据钩住，就可以像测量弹簧系数的实验一样，用第一个数据保留住以后所有的数据。由于计算机中每个变量，其本质上都是程序中的一块内存，因此其内存地址相当于钩码上的环，现在，只要为数据增加一个钩就可以解决问题了。结构体是由多个数据成员构成的，那么只要增加一个新的成员，该成员是一个指针，其类型就是该结构体的类型，那么钩的问题也得以解决，这样可以用第一个数据记录第二个数据的地址，第二个数据记录第三个数据的地址……第 n 个数据记录第 $n+1$ 个数据的地址，所有的数据可以通过第一个数据间接的访问到。这样的数据组织形式，就是链表。

利用动态空间管理，每个结构变量在堆中申请一块空间，并在结构中增加指针域指向下一块结构变量的动态空间；每一块动态空间称为一个结点。最后一个结点由于不指向任何空间，其指针域为 0(NULL)。通过第一个数据的指针可以依次访问链表中的每个结点，如图 10-5 所示。

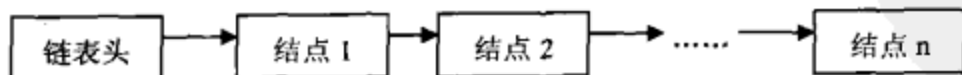


图 10-5 链表结构示意图

每个结点是结构类型，可定义为：

```

struct node
{
    T data; //T 意思是其他数据类型

```

```
    node * next  
};
```

接下来看一下创建链表并用链表保存数据和使用数据的程序。

1. 创建链表

下面的程序创建了含若干个结点的链表，结点的值依次从数组中获取。函数 creat()用于建立链表，并返回指表首的指针。函数 showlist()用于显示链表。

```
#include <iostream>  
using namespace std;  
struct node {  
    int data;  
    node * next;  
};  
void main()  
{  
    int data[5] = {1,3,5,7,9};  
    node *head=new node;    //申请首块动态空间，地址由 head 保存  
    node *p=head ;          //p 也指向表头  
    int i;  
    for (int i=1;i<5;i++)  
    {  
        p->next=new node;  
        p=p->next;  
        p->data=data[i];  
        p->next = NULL;  
    }  
    cout<<"输出结点的值是: "<<endl;  
    p = head->next;  
    while(p)  
    {  
        cout<<p->data<<endl;  
        p=p->next;  
    }  
}
```

程序运行结果如下：

```
1  
3  
5  
7  
9
```



2. 删除链表结点

要删除结点，可进行下面的操作。

- (1) 查找要删除的结点；若找不到，则返回。
- (2) p 指向待删结点，q 指向上一个结点。
- (3) q 的 next 域指向 p 的下一个结点。
- (4) 释放 p 指向的动态空间。

删除 data 域中值为 3 的结点如图 10-6 所示。

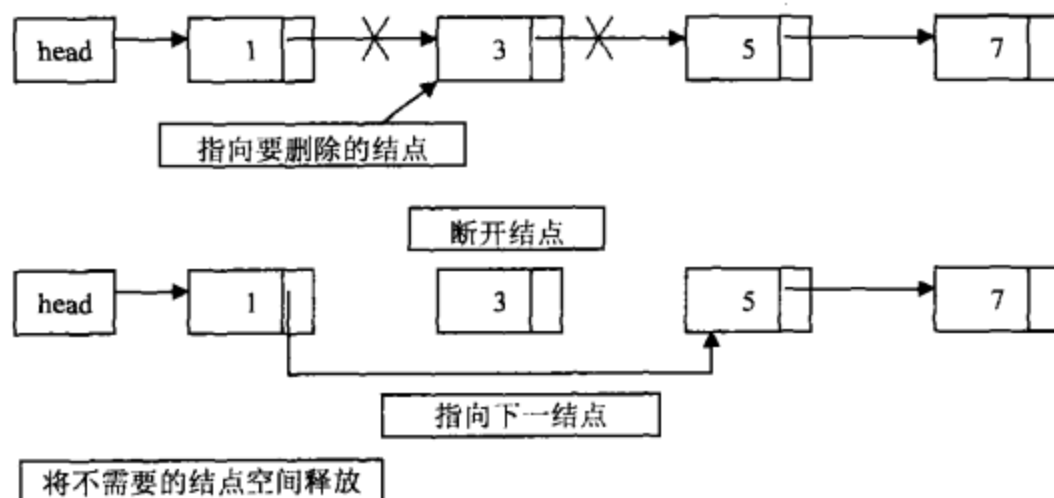


图 10-6 删除 data 域中值为 3 的结点

下面函数删除第一个结点 data 值为 num 的结点。

```
void deletenode(node * list, int num)
{
    if (!list) return;           //若链表为空，则结束
    node *p=list, *q;
    if (p->data==num)             //如果找到的结点在键首
    {
        q=p->next;
        delete p;
        return;
    }
    while(!(p->next) && p->data!=num) //查找值为 num 的结点
    {
        q=p;
        p=q->next;
    }
    if (!p) return;               //没找到，则结束
    q->next=p->next;              //q 的 next 域指向 p 的下一个结点
    delete p;                    //释放 p 指向的动态空间
}
```

3. 插入链表结点

假设链表结点按 data 域从小到大排列, 插入结点后仍然从小到大排列。

插入链表结点的过程如下。

- (1) 若链表为空, 则挂在表首指针上。
- (2) 若第一个结点的 data 值大于待插入结点的值, 则直接插在前面。
- (3) p 指向第二个结点, q 指向第一个结点, 若 p 指向的结点的 data 值小于待插入结点的值, 则 p 和 q 一直向下一结点移动(q 紧跟 p), 直到满足条件或 p 指向空。
- (4) 待插入结点的 next 指向 p 指向的结点, q 的 next 域指向待插入结点。

过程如图 10-7 所示。

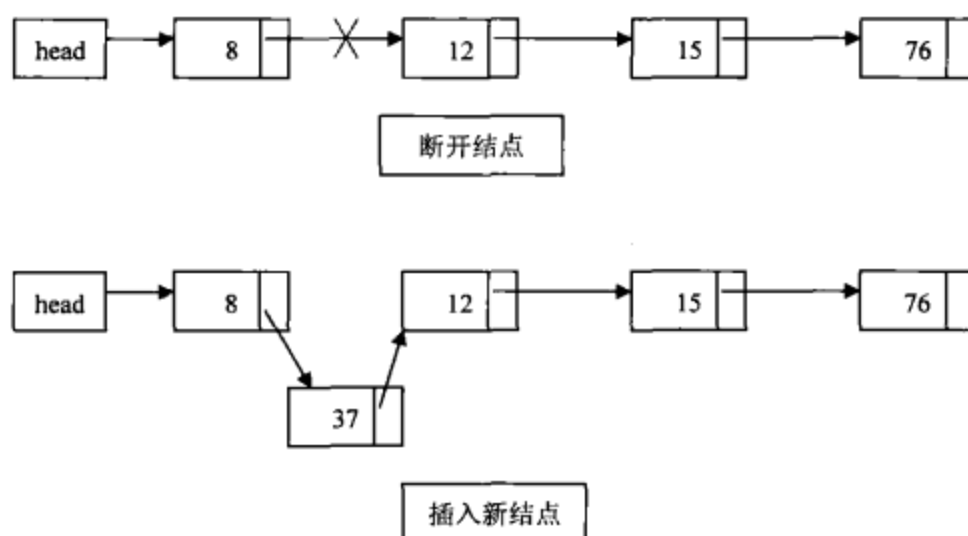


图 10-7 插入链表结点示意

按顺序插入链表结点的程序如下:

```
void insertnode(node * list,int num)
{
    if (!list)                //若链表为空, 则将结点挂在表首指针上
    {
        list=new node;
        list->data=num;
        list->next=0;
        return;
    }
    node * r=new node;        //r 为待插入结点
    r->data=num;
    if (list->data>num)        //第一个结点的 data 值大于待插入结点的值
    {
        r->next=list;
        list=r;
    }
}
```

```

        return;
    }
    node *p=list->next,*q=list; //p 指向第二个结点, q 指向第一个结点
    while (p && p->data<num)
    {
        q=p;
        p=q->next;
    }
    r->next=p;           //待插入结点的 next 指向 p 指向的结点
    q->next=r;           //q 的 next 域指向待插入结点
    return;
}

```

10.11 联 合

所谓联合类型是指将不同的数据项组织成一个整体, 它们在内存中占用同一段存储单元, 所以又称为共用体类型。其定义形式为:

```

union 共用体名
{成员表列};

```

下例表示声明一个共用体 `a_bc`:

```

union a_bc
{
    int i;
    char mm;
};

```

在用共用体类型 `a_bc` 定义的变量中, 整型量 `i` 和字符 `mm` 公用同一内存位置。共用体变量的长度为共用体中最大的变量长度。

1. 共用体变量的定义

共用体变量的定义与结构十分相似。其形式有以下三种。

(1) 在共用体声明时定义, 其形式如下。

```

union 共用体名{
    数据类型 成员名;
    数据类型 成员名;
    ...
} 共用体变量名;

```

举例如下:

```
union data
{
    int i;
    char ch;
    float f;
}a,b,c;
```

本例在声明共用体类型 **data** 的同时定义了三个 **data** 类型的变量 **a**、**b**、**c**。

(2) 先声明共用体类型，再用共用体名定义共用体变量。

举例如下：

```
union data
{
    int i;
    char ch;
    float f;
};
union data a,b,c;
```

在这个例子中，先声明了共用体类型 **data**，然后利用该类型定义了三个共用体变量 **a**、**b**、**c**。

(3) 定义无名共用体变量。

举例如下：

```
union
{
    int i;
    char ch;
    float f;
}a,b,c;
```

此例定义的共用体无名称，**a**、**b**、**c** 是无名的共用体变量。

2. 共用体成员的访问

共用体访问其成员的方法与结构体相同。

可以引用共用体变量的成员，其用法与结构体完全相同。若定义共用体类型为：

```
union data
{
    int a;
    float b;
    double c;
    char d;
}mm;
```

则其成员引用为: mm.a, mm.b, mm.c, mm.d。

但是要注意的是, 不能同时引用四个成员, 在某一时刻, 只能使用其中之一的成员。

例如, 对共用体变量的使用。

```
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
    union data
    {
        int a;
        float b;
        double c;
        char d;
    } mm;
    mm.a = 1 ;
    cout<<mm.a<<endl;
    mm.c = 1.2;
    cout<<setiosflags(ios:: fixed)<<setprecision(1)<<mm.c<<endl;
    mm.d = 'W';
    mm.b = 1.3f;
    cout<<setiosflags(ios:: fixed)<<setprecision(1)<<mm.b<<mm.d<<endl;
}
```

程序运行结果如下:

```
1
1.2
1.3
```

同样, 共用体变量也可以定义成数组或指针, 但定义为指针时, 也要用“->”符号, 此时共用体访问成员可表示成:

共用体名->成员名

另外, 共用体既可以出现在结构体内, 它的成员也可以是结构体。

例如:

```
struct{
    int age;
    char *addr;
    union{
        int i;
```

```

        char *ch;
    }x;
}y[10];

```

若要访问结构体变量 `y[1]` 中共用体 `x` 的成员 `i`, 可以写成:

```
y[1].x.i;
```

若要访问结构体变量 `y[2]` 中共用体 `x` 的字符串指针 `ch` 的第一个字符, 可写成:

```
*y[2].x.ch;
```

若写成 “`y[2].x.*ch;`” 是错误的。

3. 结构体和共用体的区别

结构体和共用体有下列区别。

(1) 结构体变量所占内存长度是各成员所占内存之和。每个成员分别占有其自己的内存单元。

(2) 共用体变量所占的内存长度等于最长的成员的长度。如上例中共用体变量 `a`、`b`、`c` 各占 4 个字节(因为一个实型变量占 4 个字节), 而不是占 $2+1+4=7$ 个字节。下面是一个相关的例子。

```

#include <iostream>
using namespace std;
union data /*共用体*/
{
    int a;
    float b;
    double c;
    char d;
};
struct stud /*结构体*/
{
    int a;
    float b;
    double c;
    char d;
};
main()
{
    cout<<"The size of the structure is "<<sizeof(struct
stud)<<"bytes"<<endl;
    cout<<"The size of the union is "<<sizeof(union data)<<"bytes"<<endl;
}

```

程序运行结果如下:

The size of the structure is 24 bytes

The size of the union is 8 bytes

由程序的输出结果可以看到, 虽然共用体和结构体的声明类似, 但是占用的内存空间却是完全不同的。共用体类型 `data` 的四个成员占用同一块内存, 占用的空间大小为其最大分量 `double` 类型的变量占用的空间(32 位机器上 8 字节), 而结构体 `stud` 占用的空间则为四个成员占用空间之和(32 位机器上 24 字节)。两者在内存中的存储见图 10-8。

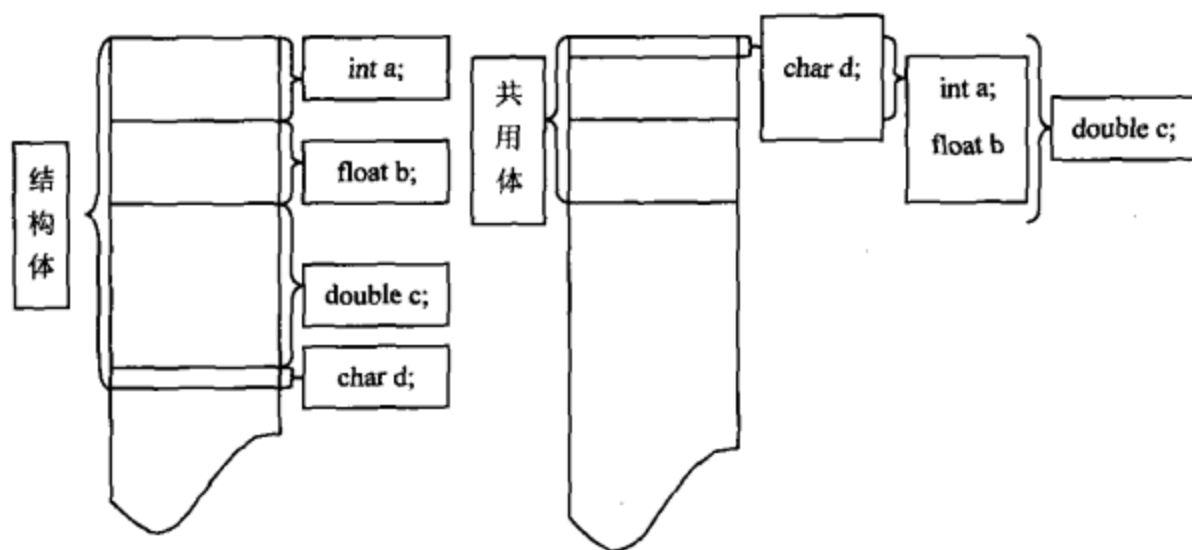


图 10-8 结构体与共用体在内存中的存储

(3) 结构体和共用体都是由多个不同的数据类型成员组成, 但在任何同一时刻, 共用体中只存放了一个被选中的成员, 而结构体的所有成员都存在。

(4) 对于共用体的不同成员赋值, 将会对其他成员重写, 原来成员的值就不存在了; 而对于结构体的不同成员赋值是互不影响的。

下面举一个例子来加深对共用体的理解。

```
#include <iostream>
#include <conio.h>
using namespace std;
void main()
{
    union{                                //定义一个共用体
        int i;
        struct{                          //在共用体中定义一个结构体
            char first;
            char second;
        }half;
    }number;
    number.i=0x4241;                      //共用体成员赋值
    cout<<number.half.first<<number.half.second<<endl;
    number.half.first='a';                //共用体中结构体成员赋值
```

```

        number.half.second='b';
        cout<<number.i<<endl;
        getch();
    }

```

输出结果为:

```

    AB
25185

```

从上例结果可以看出: 当给 `i` 赋值后, 其低八位也就是 `first` 和 `second` 的值; 当给 `first` 和 `second` 赋字符后, 这两个字符的 ASCII 码也将作为 `i` 的低 8 位和高 8 位。

4. 共用体类型数据的特点

(1) 同一个内存段可以用来存放几种不同类型的成员, 但在每一瞬时只能存放其中一种, 而不是同时存放几种。

(2) 共用体变量中起作用的成员是最后一次存放的成员。

如有以下赋值语句:

```

a.i=1;
a.ch='a';
a.f=1.5;

```

完成以上 3 个赋值语句后, 只有 `a.f` 是有效的, `a.i` 和 `a.c` 已无意义了。若用 `printf("%d",a.i)` 是不行的, 而用 `printf("%f",a.f)` 是可以的。

(3) 共用体变量的地址和它各成员的地址都是同一地址。例如: `&a`, `&a.i`, `&a.ch` 和 `&a.f` 都是同一地址值。

(4) 不能对共用体变量名赋值, 也不能企图引用变量名来得到成员的值, 又不能在定义共用体变量时对它初始化。

例如, 下面语句都是错误的。

```

union
{
    int i;
    char ch;
    float f;
}a={1,'a',1.5};    //不能初始化
a=1;                //不能对共用体变量赋值
m=a;                //不能引用共用体变量名以得到值

```

(5) 不能将共用体变量作为参数传递, 也不能使函数带回共用体变量, 但可以使用指向共用体变量的指针。

(6) 共用体类型可以出现在结构体类型定义中, 也可以定义共用体数组。反之也可, 即结构体类型和共用体类型可以嵌套使用。

10.12 枚 举

枚举是一个被命名的整型常数的集合，枚举在日常生活中很常见。

例如表示星期的 SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY 和 SATURDAY 就是一个枚举。

枚举的说明与结构体和共用体相似，其形式为：

```
enum 枚举名{  
    标识符 [=整型常数],  
    标识符 [=整型常数],  
    ...  
    标识符 [=整型常数],  
} 枚举变量;
```

如果枚举没有初始化，即省掉“=整型常数”时，则从第一个标识符开始，顺次赋给标识符 0、1、2、…；但当枚举中的某个成员赋值后，其后的成员按依次加 1 的规则确定其值。

例如下列枚举说明后，x1、x2、x3、x4 的值分别为 0、1、2、3。

```
enum string{x1, x2, x3, x4}x;
```

当定义改变成：

```
enum string  
{  
    x1,  
    x2=0,  
    x3=50,  
    x4,  
}x;
```

则 x1=0, x2=0, x3=50, x4=51。

注意：

- 枚举中每个成员(标识符)结束符是“,”，不是“;”，最后一个成员可省略“,”。
- 初始化时可以赋负数，以后的标识符仍依次加 1。
- 枚举变量只能取枚举说明结构中的某个标识符常量。

例如：

```
enum string
{
    x1=5,
    x2,
    x3,
    x4,
};
enum strig x=x3;
```

此时,枚举变量x实际上是7。

枚举类型是用户自定义的简单类型,适用于表示和处理一些非数值数据。例如,对于一个星期中的每一天、一年中的四季、人的性别、交通信号灯的三种颜色等非数值数据,若用数值类型来表示它们,比如用整数0~6分别表示星期日、星期一~星期六,用1~4分别表示春、夏、秋、冬四季,用0和1分别表示男和女,用1~3分别表示红、绿和黄三种颜色,则显得很直观,处理起来也容易出错。人们希望能用直观的方式表示这些数据,而用枚举类型就能直观地表示和处理这些数据。

1. 枚举类型的定义与初始化

1) 枚举类型的定义以及应注意的要点

枚举类型的值用标识符表示,一个枚举类型是通过列举出其所有的值来定义的。枚举的说明与结构体和共用体相似,其形式为:

```
enum 枚举名{
    标识符[=整型常数],
    标识符[=整型常数],
    ...
    标识符[=整型常数],
};
```

以下示例中分别定义枚举类型表示一个星期中的每一天,一年中的四季,人的性别,交通信号灯红、绿、黄三种颜色。

```
enum week{
    mon = 1,
    tue = 2,
    wed = 3,
    thu = 4,
    fri = 5,
    sat = 6,
    sun = 7,
};
enum season{
```

```

    spring, summer, autumn, winter
};
enum sex{male, female};
enum signal{red, green, yellow};

```

这个例子中定义了四个枚举类型，其中枚举类型 `week` 有七个元素：`sun`、`mon`、`...`、`sat`，分别表示星期日、星期一、`...`、星期六；枚举类型 `season` 有四个元素，分别表示春、夏、秋、冬四季；枚举类型 `sex` 表示人的性别男和女，枚举类型 `signal` 则表示交通信号灯红、绿、黄三种颜色。

其中 `sun`、`mon`、`...`、`sat` 等称为枚举元素或枚举常量。它们是用用户定义的标识符。这些标识符并不自动地代表什么含义。例如，不因为写成 `sun`，就自动代表“星期天”。不写 `sun` 而写成 `sunday` 也可以。用什么标识符代表什么含义，完全由程序员决定，并在程序中做相应处理。

相对于直接用数字常量，用枚举类型更直观，因为枚举元素都选用了令人“见名知意”的标识符，而且枚举变量的值限制在定义时规定的几个枚举元素范围内，如果赋予它一个其他的值，就会出现错误信息，便于检查。

定义枚举类型时应注意以下几点：

- (1) 作为枚举类型元素的标识符必须是合法的 C++ 标识符，而且不能用保留字和标准标识符，以免编译程序时出错或引起混乱。
- (2) 枚举类型元素只能用标识符表示，而不能用数值常量等非标识符的符号表示。
- (3) 在同一个分程序中，作为枚举类型元素的标识符不允许双重定义。

例如：

```

enum signal{red, green, yellow};
enum color{white, black, yellow, blue, brown};
enum green{g1, g2, g3};

```

上面的类型定义中标识符 `yellow` 既被定义为枚举类型 `signal` 的元素，又被定义为枚举类型 `color` 的元素，这种双重定义是不允许的，编译时将出错。同理，标识符 `green` 也不可既作为枚举类型 `signal` 的元素，又作为另一个枚举类型名。

- (4) 枚举中每个成员(标识符)结束符是“`,`”，不是“`;`”，最后一个成员可省略“`,`”。

2) 枚举类型的初始化以及应注意的要点

如果枚举没有初始化，即省掉“`=整型常数`”时，则从第一个标识符开始，顺次赋给标识符 0、1、2、`...`。但当枚举中的某个成员赋值后，其后的成员按依次加 1 的规则确定其值。

例如下列枚举说明后，`x1`，`x2`，`x3`，`x4` 的值分别为 0、1、2、3。

```
enum en{x1, x2, x3, x4};
```

当定义改变成:

```
enum en
{
    x1,
    x2=0,
    x3=50,
    x4,
};
```

则 x1=0, x2=0, x3=50, x4=51。

注意:

(1) 初始化时可以赋负数, 以后的标识符仍依次加 1。以下是一个相关的例子。

```
enum enum_2
{
    M_1=-5,
    M_2,
    M_3,
    M_4=10
};
void main()
{
    cout<<"M_1 = "<<M_1<<endl;
    cout<<"M_2 = "<<M_2<<endl;
    cout<<"M_3 = "<<M_3<<endl;
    cout<<"M_4 = "<<M_4<<endl;
}
```

程序运行结果如下:

```
M_1 = -5
M_2 = -4
M_3 = -3
M_4 = 10
```

(2) 枚举类型的成员为常量值, 不能对其赋值。

例如: 若对于上例添加语句:

```
M_1 = 3;
```

则程序编译时会报告错误:

```
error C2440: "=" : 无法从"int"转换为"enum_2"
```



2. 枚举变量的定义和初始化

定义枚举变量的方式也有以下三种。

(1) 在定义枚举类型时一起定义，例如：

```
enum weekday
{
    sun, mon, tue, wed, thu, fri, sat
}
weekday, week_end;
```

(2) 先定义枚举类型，然后用枚举类型名定义枚举变量。定义格式为：

枚举类型名 变量名；

例如：

```
enum weekday {sun, mon, tue, wed, thu, fri, sat};
```

此例定义了一个枚举类型 `enum weekday`，可以用此类型来定义变量，如：

```
enum weekday weekday, week_end;
```

(3) 定义无名枚举变量，例如：

```
enum {
    sun, mon, tue, wed, thu, fri, sat
}
weekday, week_end;
```

3. 枚举变量的运算

枚举类型数据所能进行的运算比较有限，只能进行赋值运算和关系运算。

1) 赋值运算

枚举类型变量的取值范围由相应的枚举类型确定，即可取定义该类型时所列举的所有枚举类型元素为其值。可对一个枚举类型变量赋以其取值范围内的任何一个枚举类型元素或同类型变量的当前值。例如，设有如下类型定义和变量说明：

```
enum week
{
    sun, mon, tue, wed, thu, fri, sat
};
enum season
{
    spring, summer, autumn, winter
};
```

```
week d1, d2;  
season seasons;
```

则下面的赋值语句都是合法的:

```
d1 = mon;  
d2 = d1;  
seasons = winter;
```

对枚举类型变量赋值时应注意以下几点。

- 赋给枚举类型变量的值不允许超出该变量的取值范围。
- 不允许将一个枚举类型变量的值赋给另一个不同类型的枚举类型变量。
如对于上面的例子, 操作: `seasons = d1;` 是非法的。
- 应区分枚举类型变量名和枚举类型元素名, 两者虽然都用标识符表示, 但有着本质上的不同, 后者表示前者的值。注意枚举类型元素不可出现在赋值号的左边。
- 只能把枚举值赋予枚举变量, 不能把元素的数值直接赋予枚举变量。如: `a=sum;b=mon;` 是正确的。而: `a=0;b=1;` 是错误的。如一定要把数值赋予枚举变量, 则必须用强制类型转换, 如: `a=(enum weekday)2;` 其意义是将顺序号为 2 的枚举元素赋予枚举变量 `a`, 相当于: `a=tue;`, 还应该说明的是枚举元素不是字符常量也不是字符串常量, 使用时不要加单、双引号。

2) 关系运算

枚举类型数据的关系运算可用于条件语句和循环语句中来表示条件。对枚举类型数据可进行六种关系运算: `=`, `!=`, `<`, `<=`, `>` 和 `>=`。

例如:

```
weekday1, day2;  
season s;
```

则下列关系表达式都是合法的:

```
sun<=tue  
summer!=spring  
day1>tue  
day1=day2  
winter<s
```

若变量 `day1`、`day2` 和 `s` 的当前值分别是 `wed`、`fri` 和 `autumn`, 则上面三个关系表达式的值为 `True`, 而后两个关系表达式的值为 `False`。



注意: 不同枚举类型的数据不可比较。例如, 下列关系表达式都是非法的:


```
mon<spring
day1>winter
```

4. 枚举的应用

以下示例先定义了枚举类型 `Season`，然后按顺序依次输出数字及对应的春、夏、秋、冬四季的英文(`spring`、`summer`、`autumn`、`winter`)。

```
#include <iostream>
using namespace std;

enum season
{
    spring, summer, autumn, winter
};

main()
{
    season month[31], j;
    int i;
    j=spring;
    for(i=1; i<=30; i++)
    {
        month[i] = j;
        j = (season)(j+1);
        if (j>winter)
            j = spring;
    }
    for(i=1; i<=30; i++)
    {
        switch(month[i])
        {
            case spring: cout<<i<<" spring"<<'\\t'; break;
            case summer: cout<<i<<" summer"<<'\\t'; break;
            case autumn: cout<<i<<" autumn"<<'\\t'; break;
            case winter: cout<<i<<" winter"<<'\\t'; break;
            default: break;
        }
    }
    cout<<endl;
}
```

程序运行结果如下：

1 spring	2 summer	3 autumn	4 winter	5 spring
6 summer	7 autumn	8 winter	9 spring	10 summer
11 autumn	12 winter	13 spring	14 summer	15 autumn

16 winter	17 spring	18 summer	19 autumn	20 winter
21 spring	22 summer	23 autumn	24 winter	25 spring
26 summer	27 autumn	28 winter	29 spring	30 summer

下面编写一个程序，输入今天是星期几，计算并输出今天、昨天和明天分别是星期几。

```
#include <iostream>
using namespace std;

enum week
{
    sun, mon, tue, wed, thu, fri, sat
};

void main()
{
    week today, yesterday, tomorrow, day;
    int i, code;
    cout<<"0 sun, 1 mon, 2 tue, 3 wed, 4 thu, 5 fri, 6 sat"<<endl;
    cout<<"请输入今天的代码: ";
    cin>>code;

    switch(code)
    {
        case 0: today = sun;
        case 1: today = mon;
        case 2: today = tue;
        case 3: today = wed;
        case 4: today = thu;
        case 5: today = fri;
        case 6: today = sat;
    }

    if(today==sun)
    {
        yesterday = sat;
    }
    else
    {
        yesterday = (week)(today-1);
    }
    if(today==sat)
    {
        tomorrow = sun;
    }
    else
    {
        tomorrow = (week)(today+1);
    }
}
```

```
for(i=1;i<4;i++)
{
    switch(i)
    {
        case 1:
        {
            day = today;
            cout<<"Today : ";
            break;
        }
        case 2:
        {
            day = yesterday;
            cout<<"Yesterday : ";
            break;
        }
        case 3:
        {
            day = tomorrow;
            cout<<"Tomorrow : ";
            break;
        }
    }
    switch(day)
    {
        case sun:
        {
            cout<<"Sunday";
            break;
        }
        case mon:
        {
            cout<<"Monday";
            break;
        }
        case tue:
        {
            cout<<"Tuesday";
            break;
        }
        case wed:
        {
            cout<<"Wednesday";
            break;
        }
        case thu:
        {
            cout<<"Thursday";
            break;
        }
    }
}
```

蘇子知覺
PDG

```
        }  
        case fri:  
        {  
            cout<<"Friday";  
            break;  
        }  
        case sat:  
        {  
            cout<<"Saturday";  
            break;  
        }  
    }  
    cout<<endl;  
}  
}
```

程序运行情况如下:

```
0 sun,1 mon,2 tue,3 wed,4 thu,5 fri,6 sat  
请输入今天的代码: 6  
Today : saturday  
Yesterday : friday  
Tomorrow : Sunday
```

本章小结

本章讲解 C++中重要的扩展类型(自定义类型)、结构体和共用体以及枚举,详细解释了自定义类型的定义和使用技巧,利用自定义类型来改善程序结构,同时讲述了链表等稍微复杂一些的数据结构相关知识。

习 题

1. 已知学生信息包括:姓名、年龄、班级、学号和成绩,自定义结构体,存储5名学生的对应信息,并按照成绩进行由大到小的排序。

2. 对于下列程序:

```
union a  
{  
    int x;  
    float y;
```

```
    char z;  
    double i;  
};  
struct b  
{  
    union a x;  
    float y;  
    char z;  
    double i;  
};
```

sizeof(b)的结果是什么?

3. 利用结构体, 实现链表的创建、排序和数据查找程序。



第 11 章 类 与 对 象

本章内容：

- 类和对象的概念。
- 类的结构和定义，对象的声明。
- 类成员的访问控制。
- 对象的空间分配。
- 构造函数、析构函数与拷贝构造函数。

重点：

- 构造函数。
- 析构函数与拷贝构造函数。
- 内存管理。

目的：

掌握面向对象编程的最基本技巧，理解类的设计基本原则，掌握数据的抽象与信息隐藏的关键技术。

从这一章开始学习的重点转移到面向对象程序设计中最基础的概念，即类与对象的概念，它们是面向对象程序设计的基本元素。

11.1 抽 象 概 述

对象乃是面向对象语言最重要的部分，对象也是面向对象语言中构成整个程序的主要成员。如果使用面向对象的设计原则来编写程序，不应该把重心摆在如何将一个程序分解成哪些函数，而是如何把一个程序分解成一组对象的总和。如果秉持着这种思想，编写程序将是一件相当轻松愉快的事，因为程序语言中的对象与现实生活中的对象有着紧密的相似性。

在面向对象程序语言中，哪些东西有资格成为对象呢？其实并没有一个明确的答案，

要根据实际情况进行处理。下列生活中的对象，可以帮助思索这个问题。

生活中真实的对象：

实物——太阳、地球、鼠标、键盘。

计算机软件——用户操作界面、窗口、按钮。

人——雇员、经理、顾客、推销员。

数据——字典、人事档案、库存资料。

以上所列出的都是现实存在的对象，从无生命的物体到有生命的人，从看得见摸得着的实物到无形的数据。

但是，在 C++ 语言中，对象要经过抽象，从而用 C++ 语言加以描述。对象首先具有它自身的数据，这些数据描述了现实对象需要被关注的方面的状态。这些数据在类中通常被叫做属性。

对象除了具备描述状态的数据外，还必须具备一些行为。也就是说，对象是一个主动的物体，只需要告诉对象该做什么即可，不必经过设计师加以操作即可完成指定的工作。这些行为在类中通常被叫做方法。

面向对象的概念将面向过程中的函数与数据合而为一，让程序中的数据与数据处理方式，与生活中的种种情形结合得更加紧密。为了在计算机程序中构建出这样的对象，需要经过抽象对现实的对象加以分析、选择。

从文字上理解，抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题，而只是选择其中的一部分，暂时不用部分细节。比如，要设计一个学生成绩管理系统，在考查学生这个对象时，其实只关心他的班级、学号、成绩等，而不用去关心他的身高、体重这些信息。

抽象包括两个方面：一方面是过程抽象，另一方面是数据抽象。过程抽象是指任何一个明确定义功能的操作都可被使用者看作单个的实体看待，尽管这个操作实际上可能由一系列更低级的操作来完成。数据抽象定义了数据类型和施加于该类型对象上的操作，并限定了对象的值只能通过使用这些操作修改和观察。

11.2 类的概念

面向对象的方法中最基本的概念之一就是类(Class)，它是对 C 语言的数据类型——结构的自然扩展。类的使用同结构非常相似，唯一不同的就是结构没有定义所说的“数据相关的操作”，“数据相关的操作”指的就是前面提到的“方法”，因此，类具有更高的抽象性。

不仅如此，类中的数据还具有隐藏性和封装性。

虽然 C++ 的类概念来自“自定义数据结构”，但是 C++ 的类却优于结构，因为类与结构一样具有方便使用的特性，而且兼具所有面向对象的优点。其实，真正的对象与类概念是创建在某些知识基础之上的技术，像是数据封装、继承、数据隐藏等，而结构并不具有以上任一种特征，因此结构并不能与类相提并论。

在真实世界中，许多对象都会具有相同的性质。例如每一个人都只是世界上所有人类的其中之一，都具有所有人类共同的特性，但是每个人又不会完全相同。

在 C++ 中，把具有相同的内部存储结构和具有相同的一组操作的对象看成是同一类的。于是，在面向对象的世界中，“人类”就是一种“类”，可以看作是一个模板；而每个人，则称为这个类下的一个“实例”(instance)。因此类可以被定义成：“类”是所有相似对象的状态变量与行为所构成的模板(template)或原型(prototype)。

换句话说，类像是一个制作模型的模子，利用这个模子可以做出许多相似但独立的对象。使用“类”，毋庸置疑地可以获得许多好处，像是模块化、软件芯片的制造，以及重复使用等，都是面向对象所带来的便利性。

类描述了一组有相同属性(数据成员 Data Members)和相同行为(成员函数 Member Functions)的对象。它将数据和作用在这些数据上的运算组合在一起，是 C++ 中提供可复用性的基本单元。

类是一种复杂的数据类型，它是将不同类型的数据和与这些数据相关的运算封装在一起的集合体。类的结构(也即类的组成)用来确定一类对象的行为，而这些行为是通过类的内部数据结构和相关的操作来确定的。对象的行为通过一种操作接口来描述(也即平时所看到的类的成员函数)，使用者只关心的是接口的功能(也就是类的各个成员函数的功能)，对它内部是如何实现的并不感兴趣。而操作接口又被称为这类对象向其他对象所提供的服务。

11.3 类的定义

11.3.1 类与结构

在 C++ 语言中，“class”关键字是专门保留给类使用的，就像是结构的“struct”关键字一样，用来定义、创建类。在 C++ 语言中，“class”与“struct”关键字，除了成员的访问控制有所差异之外，其他的特性则完全相同。假设要定义一个用于描述网络游戏中的玩家类，则可以通过下列的语句完成。


```

class player
{
public:
    //数据成员
    int playerID;           //玩家标示
    char playername;        //名字
    void walk(int x,int y);  //成员函数
};

```

是不是与结构十分神似，但是又有些许的不同呢？是的，定义一个类，就相当于定义一个全新的数据类型，与结构的概念是一模一样的。

上述类中还多了个“public:”语句，其后的变量就是该类的数据成员，它们用于存储数据，而函数 walk() 是成员函数，表示了玩家可能的行为。在以后的篇幅内将分别详细讲解类的数据成员和成员函数。

类与结构体的区别除关键字不同外(class 或 struct)，唯一的区别是，结构在默认情况下的成员是公共的，而类在默认情况下的成员是私有的。因此结构体的所有其他特性都适用于类，例如结构体的引用、结构体指针等。

11.3.2 类的声明

类定义包含以下两部分。

- 类头(class head)由关键字 class 及其后面的类名构成，类头本身也用作类的声明。
- 类体(class body)由一对花括号包围起来，类定义后面必须接一个分号或一系列声明来结尾。类体包含成员定义以及访问控制标签，如 public 和 private 等。类的成员包括该类能执行的操作和代表类抽象所必需的数据，这些操作称为成员函数或方法。而类中的数据，称作成员变量或属性。

类的定义格式如下：

```

class 类名标识符      //类头
{ //类体，包括数据成员和成员函数
private:              //访问控制标签
    //...
protected:
    //...
public:
    //...
};

```

例如:

```
class myclass           //类头, 类名为 myclass
{
public:
    void display();      //成员函数
private:
    int data;            //数据成员
};                       //以分号结尾
```

C++允许在定义类的同时声明这个类的对象, 例如:

```
class myclass
{
    //...
}object1,object2;       //同时声明了两个 myclass 对象 object1 和 object2
```

要注意的是, 每个类定义引入一个不同的类类型。即使两个类类型具有完全相同的成员表, 仍然认为它们是不同的类型。例如:

```
class myclass
{
public:
    void display();
private:
    int data;
};
```

和

```
class yourclass
{
public:
    void display();
private:
    int data;
};
```

是不同的类型。

11.3.3 类成员的访问控制

1. 封装

封装是 C++ 面向对象的第一大特性。封装就是指 C++ 中类的私有变量只能由该类的成员函数引用, 非成员函数或其他类的成员函数都不能直接引用的机制。

C++的封装特性使得软件的维护更加方便。修改内部的数据结构和函数的实现不会影响其他软件成分的实现。封装所造成的信息隐藏减少了软件成分之间的相互依赖，因而大大地减少了程序的复杂性，提高了可靠性和可修改性。

封装是为了防止程序的函数直接访问类类型的内部表示而提供的一种形式化机制。类成员的访问限制是通过类体内被标记为 `public`、`private` 以及 `protected` 的部分来指定的。关键字 `public`、`private` 和 `protected` 被称为访问限定符(`Access Specifier`)。在公有区(`public`)内被声明的成员是公有成员；在私有区(`private`)或被保护区(`protected`)域内被声明的成员是私有或被保护的成员。

公有成员在程序的任何地方都可以被访问。实行信息隐藏的类将其公有成员限制为成员函数，这种函数定义了可以被一般程序用来操纵该类类型对象的操作。因为使用 `public` 的类成员是可以被其他程序所访问的，就像是结构的成员一般。但是，那些没有设置成“`public`”的类成员，其他的程序并没有权力加以访问，也就是说，类内的变量数据是可以被加以保护的，以防止其他程序擅改数据。如果省略“`public`”关键字，则 C++ 会将成员变量默认成“`private`(私有的)”，也就是说，除了该类内的程序之外，其他程序不能对这些变量加以更改。

私有成员只能被成员函数和类的友元访问，实行信息隐藏的类把其数据成员声明为 `private` 的。

保护成员对类本身的派生类(`Derived Class`)就像公有成员一样，对其他程序代码则表现得像私有成员。关于保护成员的完全讨论将会在第 12 章进行，在介绍继承以及派生类的概念时读者将看到怎样使用保护成员的实例。

封装是软件工程中一个非常重要的概念，在后面的章节中将为您详细地介绍，现简要说明它为程序提供的两个主要好处。

- 如果类的私有实现代码需要修改或扩展，那么只有相对很小一部分要求访问这些实现代码的成员函数需要修改，而许多使用该类的用户程序无须修改，只是要求重新编译而已。
- 如果类的私有实现代码有错误，那么通常需要检查的代码数量只局限在相对较少的需要访问这些实现代码的成员函数上，而无须检查整个程序。

“`public`”和“`protected`”(或“`private`”)^①关键字的使用，对于类内的数据有相当大的保护作用，这就是面向对象语言的第一大特性：封装性。适时地使用“`protected`”(或“`private`”)

① 有关 `protected` 与 `private` 的区别，参见第 13 章“继承与多态”。

关键字可以有效避免由于其他程序篡改数据导致类的运行不正常。同时在使用类时，也可以不必理会那些“protected”（或“private”）的成员，只要知道“public”成员如何运行即可。

2. 私有与公有成员函数

类的成员函数可以被声明在类体的公有、保护和私有区内。怎样判断一个成员函数应该被放在哪儿呢？

公有成员函数是用于定义类的用户可能想执行的操作，公有函数集定义了类的对外接口集合。例如类 `Screen` 的成员函数 `home()`、`move()` 和 `get()` 定义了可被程序用来操纵 `Screen` 型对象的操作。因为通过把数据成员定义为私有的，以便向类的用户隐藏类的内部表示，所以必须提供公有成员函数来操纵 `Screen` 对象，这就是封装。

封装使得代码免受因某一个类内部实现的变化而带来的影响，同时也防止了类对象的内部数据成员被程序任意地修改。而使用一个较小的公有函数集——接口，提供了对该类对象的全部修改动作。如果出现错误，则错误的查找空间局限在这个函数集中，这大大地简化了程序的维护和修正问题。

到目前为止只看到支持读取私有数据的成员函数，如下面有两个 `set()` 函数，它们允许用户修改 `Screen` 对象，这些成员函数的定义如下。

```
class Screen
{
public:
    void set( const string &s );
    void set( char ch );
};

void Screen::set( const string &s )
{
    // 在当前 _cursor 位置写字符串
    int space = remainingSpace();
    int len = s.size();
    if ( space < len )
    {
        cerr << "Screen: warning: truncation: "
              << "space: " << space
              << "string length: " << len << endl;
        len = space;
    }
    _screen.replace( _cursor, len, s );
    _cursor += len - 1;
}

void Screen::set( char ch )
```



```

{
    if ( ch == '\0' )
        cerr << "Screen: warning: "
        << "null character (ignored).\n";
    else
        _screen[_cursor] = ch;
}

```

Screen 类实现假定 **Screen** 对象不包含内嵌的空字符, 这是 **set()** 不允许向 **Screen** 写空字符的原因。给出的函数是公有成员函数, 它们可以在程序的任何位置上被调用。但是私有成员函数只能被类的其他成员函数和友元调用, 程序不能直接调用它们。在实现类的功能时, 私有成员函数为其他成员函数提供支持。如在函数 **set(const, string&)** 中用到的函数 **remainingSpace()** 就是这些函数之一, 它是类 **Screen** 的私有成员函数, 用以返回屏幕上剩余的空间数, 其实现方式如下。

```

class Screen
{
public:
    // 其他的成员函数声明保持不变
private:
    inline int remainingSpace();
};
inline int Screen::remainingSpace()
{
    // 当前位置不再是剩余的
    int sz = _width * _height;
    return( sz - _cursor );
}

```

11.3.4 数据成员

数据成员又叫做属性, 它定义了这个类的对象所拥有的数据。数据成员的声明方式与普通变量完全相同, 唯一区别在于普通变量声明在全局或局部作用域, 而类的数据成员声明在本身的类域之中(类域的概念接下来就要讲到)。

数据成员可以是 C++ 所使用的所有类型, 包括基本数据类型、数组、指针、用户自定义数据类型(如枚举、结构), 以至类类型本身。例如:

```

class c1
{
public:
    int a;
    char * b;
    int c[10];
}

```

```
        //...
protected:
    c2 Info;
};
class c2
{
    //...
}
```

这里假设类(class c1)的数据成员有包括整型和字符串,用自定义类型 class c2 记录其他自定义类型的数据。

11.3.5 成员函数

类的成员函数是一组操作的集合,用户可以在该类的对象上执行这些操作。它们定义了对象可能的行为。

相对于数据成员,类成员函数的定义有些特殊。回顾一下,函数的定义包括函数声明和函数体定义,这两部分是可以分开的。而类的声明和定义也是可以分别定义的,因此成员函数的声明就有了两种方式:在类体内的内联定义,或是非内联的定义(常规定义)。

1. 内联成员函数定义

内联成员函数定义有两种形式:隐式和显式定义。

对于隐式定义,其成员函数的声明和定义是在类体内提供的,它们被称为在类体中隐式定义的内联函数。即使前面没有加上 `inline` 关键字,这些函数也将被编译器自动作为内联函数处理。例如:

```
class c1
{
public:
    void display()
    {
        cout<<"data..."<<data;
    }
protected:
    int data;
};
```

这里的 `display()` 函数自动被作为内联函数处理。

也可以明确地标明成员函数的内联性质,程序如下,但它的效果和上面的例子是完全一样的,函数 `display()` 都会被作为内联函数,关键字 `inline` 实际上是多余的,因此在以后的

类的定义中将不再写出 `inline` 关键字。

```
class c1
{
public:
    inline void display()
    {
        cout<<"data...s "<<data;
    }
protected:
    int data;
};
```

而对于显式定义，其函数则不是在类体内定义。和常规函数的定义方法一样，将函数声明与定义分开，在它们前面都加上关键字 `inline`，明确声明成员函数为内联的。例如：

```
class c1
{
public:
    inline void display()
protected:
    int data;
};
inline void c1::display()
{
    cout<<"data..."<<data;
}
```

要注意的是，在这种方式下，函数的声明和定义前都必须加上 `inline` 关键字。

成员函数 `display()` 仅仅是用来在屏幕上显示类唯一的数据成员的值，只有一句输出语句。所有在类体中定义的成员函数既然是内联的，就需要遵守内联函数的定义规则：规模较小，操作简单，语句少(1~5 句)，不能包含循环和 `switch` 语句。否则，即使有 `inline` 说明符，编译器还是无法把它作为内联函数来处理。

2. 常规定义

另一种方法，也是类成员函数的一般定义方式，即：在类体内进行声明，在类体外进行定义。这种方法对应于较复杂的成员函数，如包含循环和 `switch` 语句，规模大，还有就是不适于在类体中定义的成员函数。

在类体外的函数的定义语法格式为：

```
返回类型 类名 :: 成员函数名(形参列表)
{
    //...函数实现
```

```
}
```

其中的“::”叫做作用域区分符。由于域的存在，不同作用域的对象可以同名，在一个作用域内使用另一个域的对象时，应使用域名加上作用域区分符来修饰对象，以表示它是属于另一个域的。这样也可以使得同名对象可以用作用域区分符加以区分。例如下面例子中的 init()函数。

```
class c1
{
public:
    inline void display()
    {
        cout<<"data... "<<data;
    }
    void init();
protected:
    int data;
    int array[10];
};
void c1::init()
{
    for(int i=0;i<10,i++)
    {
        array[i]=i;
        cout<<array[i]<<" ";
    }
}
```

这是最为正规而常用的定义方式，对于开发大型的软件，通常把类的定义和其成员函数的定义分开进行。

在实际开发工作中，protected 类型的成员是比较常见的，而除非有特殊的需要，才会声明 private 类型的成员。

11.3.6 重载成员函数

同一个类的成员函数之间可以形成重载，方法与非成员函数重载相同。由于类名是成员函数名的一部分，所以不同类的成员函数同名，不是函数重载，类的成员函数与非成员函数同名，也不是函数重载。例如，下面的代码中定义了 Student 类的两个重载函数，定义了 Slope 类的一个成员函数，定义了一个普通函数，并在主函数中分别调用了它们。

```
class c1
{
public:
```



```

float func(int x, int y){//...}    //以两个坐标值参数
float func(Point p)
{
    ::func2(p);                    //调用全局函数 func1(Point)
    //...
}                                    //以一个描述位置的点结构为参数
//...
protected:
    //...
};
void func2(Point p)
{
    //...
}
void main()
{
    Point p(110,220);              //初始化点结构
    cl a;
    a.func(p);                     //调用 cl::func(Point)
    a.func(103,220);               //调用 cl::func(int,int)
    func2(p);
}

```

在主函数运行时，一共调用了三个重载函数，分别匹配类中重载的成员函数，重载的全局函数。在类成员函数内也调用了全局函数，域区分符“::”没有跟在类名后时表示引用的是全局变量或函数。

11.3.7 类定义的注意事项

在定义类时需要注意以下几点。

(1) 在类体中不允许对所定义的数据成员进行初始化。例如：

```

class cl
{
    protected:
        int data=100;    //错误
}

```

在类定义中的赋值是不允许的，因为类的定义是一种类型的说明，它是没有值的，只有类型的实例，相对于普通类型而言是变量，相对于类来说就是“对象”，是有特定的值的。

(2) 类中数据成员的类型可以是任意的，包含整型、浮点型、字符型、数组、指针和

引用等，也可以是对象。另一个类的对象，可以作该类的成员，但是自身类的对象是不可以的。因为当类被定义时，它的结构还不明确，无法得到它自身的对象大小。而一个指向自身类的指针或引用则是可以的。

根据 C++ 的提前说明规则，当一个类需要另一个类的指针或引用作为它的成员时，如果在这之前没有看到另一个类的定义，则需要提前说明。例如：

```
class B;           //类声明
class A
{
private:
    B *b;          //B 的定义在指针引用之后
}
class B            //类定义
{
    //...
}
```

(3) 一般的，在类体内先说明公有成员，它们是用户所关心的，后说明私有成员，它们是用户不感兴趣的。在说明数据成员时，一般按数据成员的类型大小，由小至大说明，这样可提高空间利用率。

(4) 应习惯地将类定义的说明部分或者整个定义部分(包含实现部分)放到一个头文件中。

11.3.8 类声明和类定义

前面说过，类声明和类定义是不同的，下面来详细比较一下它们。一旦到了类体的结尾，即结束处的右括号，就称一个类被定义了一次。一旦定义了一个类，则该类的所有成员就都是已知的，类对象的大小也是已知的了。也可以声明一个类但是并不定义它，例如：

```
class people;      //people 类的声明
```

这个声明向程序引入了一个名字“people”，指示“people”为一个类类型，但是只能以有限的方式使用已经被声明但还没有被定义的类型。如果没有定义类，那么就不能定义这类类型的对象，因为类类型的大小不知道，编译器不知道要为这种类型的对象预留多少存储空间。但是可以声明指向该类类型的指针或引用，允许指针和引用是因为它们都有固定的大小，这与它们指向的对象的大小无关。

但是，因为该类的大小和类成员都是未知的，所以要等到完全定义了该类，才能将解引用操作符“*”应用在这样的指针上，或者使用指针或引用来指向某一个类成员。只有已经看到了一个类的定义，才能把一个数据成员声明成该类的对象，若在程序文本中还没有看到该类定义的地方，数据成员只能是该类类型的指针或引用。例如，下面是类 department

的定义，它有一个数据成员是指向 `people` 类的指针，但 `people` 类仍未定义。这里 `people` 需要被提前声明。

```
class people;           // 声明
class department
{
    people *worker;      // ok: 指向一个 Screen 对象
    ...;
};
```

11.4 对 象

11.4.1 类与对象的区别和联系

类与对象的关系如同基本数据类型与普通变量的关系，如定义变量：

```
int j, k;               //int 是类型，j,k 是 int 类型的两个对象或称实例
```

定义了一个类，并不占据存储空间，而只是存在了一种用户可以用以声明变量的类型；而定义了一个类的对象，该对象拥有数据成员，它们占据一定的存储空间。类的所有成员（数据成员或成员函数）必须通过类的对象进行访问，类的成员函数必须通过类的对象进行调用。

11.4.2 对象的声明

类在 C++ 程序中是无法直接使用的，之前说过，类只相当于模板，而模板并不是实际可用的物品，因此在要使用类的成员时，必须先创建一个属于该类的“对象”。

声明对象的方式如同声明其他数据类型变量或者结构一样，一旦定义好类之后，就能合法声明此类的变量，声明的方式有以下两种。

- 在类定义的同时声明，如：

```
class c1
{
    //...
}obj1,obj2;
```

- 或者更简单的，像普通变量一样声明，如：

```
c1 obj1,obj2,
c1 *po=& obj2;
```

11.4.3 访问数据成员

在类体中定义了一个新的域——类域。在类体中每一个类成员的声明都向它的类域中引入了一个成员名。

通过使用两个成员访问操作符(member access operator), 可以通过名字调用类的数据成员和成员函数。这两个操作符分别为用于类对象的点操作符“.”, 以及用于类对象指针的箭头操作符“->”。

成员访问操作符“.”和“->”可以被用在程序中来访问类域中声明的成员。使用“.”和“->”操作符时在操作符前面的名字给出了一个类类型的对象或指向一个类对象的指针。对于操作符后面的成员名, 编译器将在这个类的类域中查找。例如:

```
c1 obj1,*obj2;           //声明类对象与类对象的指针
obj2=&obj1;               //初始化类指针
obj1.data=1;
obj2->data++;
cout<<obj2->data<<endl;
```

11.4.4 调用成员函数

一个对象的行为, 是通过调用它的成员函数来表现的。调用成员函数的形式类似于访问对象的数据成员。调用时必须指定对象和成员名, 否则无意义。例如, 下面的代码错误地使用类名来调用成员。

```
class c1
{
public:
    float func(int x, int y){//...}    //以两个坐标值为参数
    float func(Point p)
    {

    }
protected:
    int data;
};
c1 s; //全局对象名为 s
void func()
{
    c1::data =10;           //错误: data 是一个数据成员, 不能确定它属于 c1 类的哪个对象
    c1::func(15, 1000);    //错误: 不能确定 func 对哪个对象操作
}
```

下面的代码才是正确的成员函数调用形式。

```
void test()                //普通函数
{
    cl a;                  //创建对象
    a.func(15, 1000);      //调用其成员函数
}
```

成员函数也可以由指针来引导。例如：

```
void main()
{
    cl a, *b;
    b = &a;
    b->func(10,20);
}
```

还可以用引用传递来访问成员函数。用对象的引用来调用成员函数，看上去和使用对象自身的简单情况一样。

例如：

```
void main()
{
    cl a;
    cl &b = a;
}
```

成员函数必须用对象来调用。另一方面，在成员函数内部，访问数据成员或成员函数则无须如此。

例如，下面的程序中，成员函数内部访问了数据成员。

```
class cl
{
public:
    float func(int x, int y)
    {
        data1 = x;
        data2 = y;
        Display();
    }
    void Display()
    {
        cout<<data1<<" "<<data2<<endl;
    }
protected:
```



```
        int data1;
        int data2;
};

void main()
{
    cl a,b;
    a.func(10,20);
    b.func(30,40);
}
```

在主函数中, 创建了 a 和 b 对象, 然后 a 和 b 对象调用了成员函数 func()。在 func() 成员函数中访问了 data1 和 data2。

一个类中所有对象调用的成员函数都是同一代码段。那么, 成员函数又是怎么识别 data1 和 data2 是属于哪个对象的呢?

原来, 在对象调用 a.func(10,20) 时, 成员函数除了接受两个实参外, 还接受了一个对象 a 的地址。这个地址被一个隐含的形参 this 指针所获取, 它等同于执行 this=&a。所有对数据成员的访问都隐含地被加上前缀 this->。在类的成员函数内, 当函数被调用的时候, 存在一个隐含的指针, 指向调用该函数的类的对象。例如在上例中, 当对象 a 调用函数 func 时, this 指向对象 a, 而当对象 b 调用函数 func 时, this 指向对象 b。所以:

data1 = x; 等价于 this-> data1 = x; 等价于 a.data1 = x;

因此, 无论对应哪个对象调用的成员函数从获得的参数(显式的和隐含的)来判断都清楚, 这就是成员函数中访问成员无须对象名作前缀的原因。func() 成员函数还可表示成下列代码。

```
float func(int x, int y)
{
    this->data1 = x;
    this->data2 = y;
    this->Display();
}
```

但不可表示成下列代码。

```
float func(int x, int y)
{
    a.data1 = x;
    a.data2 = y;
    a.Display();
}
```

因为成员函数是所有对象共享的代码, 不是某一个对象所独占的, 所以不能在成员函

数内使用某个特定的对象。在编译期间，a 对象由于没有在成员函数内部或文件作用域中声明而导致失败。一个类对象所占据的内存空间由它的数据成员所占据的空间总和所决定。类的成员函数不占据对象的内存空间。

11.5 综合应用

本书第 8 章 8.13 节设计了一个简单的栈，在第 9 章讲述指针与引用时曾对该程序进行了改进，虽然已经形成了一个可以使用的栈，但从实际应用角度考虑，仍有很多不足，其中两个比较显著的问题如下。

- Data、Pos、Size 这三个变量，作为全局变量，任何其他函数都可以访问，那么只要有一个函数错误地设置这三个变量中的值，则栈的数据就会遭到破坏，在实际应用中后果是不确定的。
- 如果仅仅使用一个栈，程序还比较简单，但万一需要多个栈，则要么设计更多的函数，要么将上述 3 个变量都变为数组(Data 变为二维数组)，同时在每个函数中都增加一个参数，用于指定操作数组中的第几个元素，这给编程带来极大的不便，同时也使得程序的可读性很差。

为了解决这两个问题，可以用将栈的设计修改为用类的方式来描述，设计过程中，主要解决两个问题：一是保护数据，二是要方便的定义多个栈。综上所述，程序设计如下。

```
class Stack
{
public:
    bool Pop(int &t);
    bool Push(int x);
    void Init();
protected:
    int *Data;
    int Pos;
    int Size;
};

void Stack::Init()
{
    Data=new int[200];
    Pos=0;
    Size=200;
}

bool Stack::Push(int x)
```



```
{
    if(Pos==Size)
    {
        return false;
    }
    Data[Pos++]=x;
    return true;
}

bool Stack::Pop(int &t)
{
    if(pos>0)
    {
        t = Data[--Pos];
        return true;
    }
    else
    {
        return false;
    }
}
```

程序经过这样改造的好处有以下几点。

- 变量 Data, Pos, Size 变成了类的成员变量, 并且其属性为 `protected`, 这样, 在类的外部如果写成:

```
Stack a;
a.Pos = 0;
```

则无法编译通过, 因此也就不会被错误地访问, 数据的安全性得到有效保障。

- 同时, 当需要多个栈时, 只要定义多个 Stack 类的对象或者 Stack 类型的数组即可。

11.6 构造函数

11.6.1 为何需要构造函数

在平时的生活中养成一些好习惯, 生活才不会变得乱七八糟。当然编程中也不希望代码乱七八糟, 所以对于代码的编写也有一些准则, 其中一条就是小心使用未初始化的变量。在意外的情况下, 使用一个未初始化的变量简直就是灾难, 而使用一个未初始化的指针将导致崩溃。

在编程时，要记得随时初始化变量。在 C++ 中，初始化不会降低效率，要养成好习惯，声明一个变量的时候就立即将它初始化，对于对象也是一样。

假设声明了类 `Stack` 的对象 `Obj`，由于封装的存在，不可能直接对私有成员进行初始化，只可以寄希望于成员函数。例如，使用两个成员函数 `Init()` 和 `Free()` 来完成对象数据成员的初始化和清除，如：

```
Obj.Init();
```

这样的调用并不是很困难，要记住也不是难事。但是问题在于，谁都不能保证程序员永远不会忘记。例如：

```
Stack Obj;  
Obj.Push(10);
```

这样的使用方法编译器不会报错。但是，当执行 `Push` 时，由于 `Data` 还没有分配内存，因此 `Data[0]` 是没有访问权的，这是非常可怕的逻辑错误。

对于一个对象的生存时间，若对象在出生的时候被初始化，死亡的时候被释放，假如这一切能用这样的机制来操作，就再也不用担心会由于忘记或错误地使用而带来麻烦了。

在 C++ 中，初始化实在太重要了，以至于这项工作最好不要留给用户来完成。于是 C++ 的设计者提供了一个叫做构造函数的特殊函数来保证每个对象都被正确的初始化。如果一个类有构造函数，编译器在创建对象时就自动调用这一函数，这一切在用户使用他们的对象之前就已经完成了。对用户来说，是否调用构造函数并不是可选择的，它是由编译器在对象定义时自动完成的。

接下来的问题是这个函数该叫什么名字。这必须考虑两点，首先这个名字不能与类的其他成员函数冲突；其次，因为该函数是由编译器调用的，所以编译器必须总能自动知道调用哪个函数。最容易也是最符合逻辑的方法就是：构造函数的名字与类的名字一样。这使得这样的函数在初始化时自动被调用。下面是一个带构造函数的类的简单示例。

```
class X  
{  
    int I;  
public:  
    X();          //构造函数  
}
```

现在当一个对象 `a` 被定义时：

```
X a;
```

这时就好像 `a` 是一个整型变量一样，为这个对象分配内存。但是当程序执行到 `a` 的定

义点时，构造函数自动被调用，因为编译器已悄悄地在 `a` 的定义点处插入了一个 `X::X()` 的调用，就像其他成员函数被调用一样。传递到构造函数的第一个参数(隐含)是调用这一函数对象的地址。

像其他函数一样，也可以通过构造函数传递参数，指定对象该如何创建，设定对象初始值等。构造函数的参数保证对象的所有部分都被初始化成合适的值，它解决了类的很多问题，并使得代码更容易阅读。

构造函数和析构函数是两个非常特殊的函数：它们没有返回值。这与返回值为 `void` 的函数不同。后者虽然也不返回任何值，但还可以让它做点别的。而构造函数和析构函数则不允许。在程序中创建和消除一个对象的行为非常特殊，就像出生和死亡，而且总是由编译器来调用这些函数以确保它们被执行。如果它们有返回值，要么编译器必须知道如何处理返回值，要么就只能由用户自己来显式地调用构造函数与析构函数，这样一来，安全性就被破坏了。

构造函数不可以在定义对象之外的场合调用，例如：

```
X a;  
a.X(); // 错误，构造函数不能这么使用，只能由编译器调用
```

11.6.2 构造函数的定义

构造函数是类中一种特别的成员函数，当设计师定义好一个类以及构造函数时，往后只要声明该类的对象，C++就会自动调用该函数，为对象的成员进行初始化，不必再大费周章地为每个变量指定初始值，确保成员变量的内容为有效数据。以下定义一个 `Stack` 类，并为此类定义一个构造函数，为类成员指定初始值。

```
class Stack  
{  
public:  
    Stack();  
protected:  
    int *Data;  
    int Pos;  
    int Size;  
};  
  
Stack::Stack()  
{  
    Data=new int[200];  
    Pos=0;  
    Size=200;
```



```
}
```

示例程序中的构造函数 `Stack` 并没有被传入任何参数，只是单纯地为成员变量设置初始值而已。但是这并不表示构造函数不能传入任何参数，只是这个程序没有使用这个功能罢了。否则，所有的对象都会被初始化为千篇一律，构造函数也就失去了它的意义。

不能传入参数的构造函数，在对象被创建的同时只能给予其成员变量固定的初始值，这对于程序设计人员来说并不是什么好消息，因为利用该函数所创建的对象无法依照需求指定成员变量的初始值，在这种情况下，设计人员势必得多费几个步骤，将成员变量的内容指定成符合需求的数据。因此必须学习如何让构造函数可以利用外部程序所给的数据，为变量初始化。

11.6.3 带参数构造函数

C++中的构造函数并不是只能为成员变量设置固定的值，与一般的函数一样，也允许从外部传入数值，让构造函数运算并当作初始化的依据。

要让构造函数可以传入参数，必须在定义构造函数时加以明确定义传入参数的个数和数据类型，这一点与平常使用的函数是完全相同的，但是注意构造函数是无法返回任何数据的。声明具有传入参数的方式如下：

构造函数名称(形式参数表)；

如果只有一个参数，指定参数的数据类型与名称即可，若有两个以上的参数，则参数之间必须以逗号分隔，与函数的定义相同。

下面程序中，为了让用户可以自行指定栈的大小，因此将类的构造函数设计成具有传入参数的样式，并通过构造函数依据输入的数值，设置成员变量的初始值。

```
class Stack
{
public:
    Stack(int len);
protected:
    int *Data;
    int Pos;
    int Size;
};

Stack::Stack(int len)
{
    Data=new int[len];
    Pos=0;
```



```
    Size=len;
}
```

示例中，把构造函数设计成具有参数的函数，因此创建对象时，可以直接将参数传入，在创建对象的同时，一并为成员变量进行初始化。初始化可以是赋值操作，也可以是其他为对象正确创建而做的操作，如这里的 `strcpy` 函数调用。

但是，这种方式并不安全，如果设计人员忘了要传入参数，会因为成员变量没有初始化而导致程序错误。例如：

```
Stack a(100);
```

是可以的，而现在：

```
Stack a;
```

反而无法运行了。为什么会这样？在前面并没有显式的定义一个构造函数的时候，那些没有构造函数的类不是也正常的使用吗？这一切都是因为——默认构造函数(也称为缺省构造函数)。

11.6.4 默认构造函数

每个类都只有一个析构函数和一个赋值函数，但可以有多多个构造函数(包含一个拷贝构造函数，其他的称为普通构造函数)。对于任意一个类，如果没有编写上述函数，C++编译器将自动产生四个默认的构造和析构函数，如类 A：

```
A(void);           // 默认的空参数构造函数
A(const A &a);      // 默认的拷贝构造函数
~A(void);          // 默认的析构函数
A & operate =(const A &a); // 默认的赋值运算符重载(见运算重载内容)
```

这不禁让人疑惑，既然能自动生成函数，为什么还要程序员编写？原因有以下几点。

- 如果使用“默认的空参数构造函数”和“默认的析构函数”，等于放弃了自主“初始化”和“清除”的机会，实际上由编译器提供的默认构造函数与析构函数什么初始化工作也没做。
- “默认的拷贝构造函数”和“默认的赋值函数”均采用“位拷贝”而非“值拷贝”的方式来实现，倘若类中含有指针变量，这两个函数注定将出错。这一点在后面深拷贝与浅拷贝内容中会讲到。

问题在于，在 C++ 的机制中，一旦程序员为自己所编写的类提供了自定义的构造函数，编译器将不再提供默认的构造函数，这就造成了 11.6.3 节中的尴尬。因此，默认构造函数看来就是十分必要的了。

默认构造函数是指能够被无参数调用的构造函数，以下两种情况不需要参数。

- 构造函数没有参数，如：

```
Stack()
{
    Data=new int[200];
    Pos=0;
    Size=200;
}
```

- 构造函数有默认参数，使得在创建对象时可以不给出参数，如：

```
Stack(int len = 200)           //构造函数的参数有默认值，可以不用给出
{
    Data=new int[len];
    Pos=0;
    Size=len;
}
```

这两种情况都是类的默认构造函数，但是由于默认构造函数是被自动调用的，所以很明显，一个类中默认的构造函数只能有一个，否则编译器将无法判断需要调用哪一个。如果在类中没有显式定义构造函数，那么编译器会自动地隐式创建一个，这个隐式创建的构造函数就是默认构造函数。但是它和一个空的构造函数很相像，除了产生对象的实例以外什么工作都不做。例如在一个对象被声明的时候，就会引起默认构造函数的调用。因此类必须有默认构造函数，不论是自动生成的还是设计者定义的。作为一条原则：要么不定义自己的构造函数，一旦定义了构造函数，就必须同时提供默认构造函数。

11.6.5 重载构造函数

既然 11.6.4 节中的例子并非安全，是否有更安全的方法可以避免这种情况呢？简单地想，只要在类中同时使用具有参数与不具有参数的构造函数，就能避免这种错误情形的发生。构造函数的名字必定相同，又都没有返回类型，仅仅是参数不同，这正是重载构造函数的使用。

因为创建对象时，若没有传递任何参数，则 C++ 编译器会自动采用无参数的构造函数创建对象；如果传入参数，自然就会自动选用具有参数的构造函数了。

```
class Stack
{
public:
    Stack();
    Stack(int len);
protected:
```

```
    int *Data;
    int Pos;
    int Size;
};
Stack::Stack()
{
    Data=new int[200];
    Pos=0;
    Size=200;
}
Stack::Stack(int len)
{
    Data=new int[len];
    Pos=0;
    Size=len;
}
```

从这个范例中不难发现,一个类中可以依据需要设计多个构造函数,让使用此类设计程序的程序员可以按照需求创建合适的对象。假设有一个类,可以有1~3个传递参数,那么,就可以在类中分别设计3种不同的构造函数,让使用者依据需求使用。因此,类的设计者的一项重要任务就是为类完成一组恰当的构造函数,从而使使用者可以方便灵活地使用他的类来声明使用者所需的对象。

不过,无论类中有多少个构造函数,它们的名称都必须与类名称相同,只是参数的个数不同而已。

11.7 类对象成员的初始化

当定义类时,数据成员可以是普通变量,也可以是其他类的对象。对于普通变量,它的初始化仅仅是简单的赋值即可;对于对象成员,则必须提供一种机制使得在本类的构造函数被调用时,自动地去调用成员对象的构造函数。因为除了构造函数以外,没有其他方法可以初始化对象。

C++为类对象成员的初始化提供的特殊的初始化方式叫做“类成员初始化表(简称成员初始化表)”。初始化表位于成员对象所属类的构造函数原型的参数表之后,函数体的“{”开始之前。这样的安排,说明该表里说明的初始化工作发生在构造函数体内的任何代码被执行之前。其形式如下:

构造函数原型(参数表):成员对象1(初始化参数表),成员对象2(初始化参数表)...\n{...//函数体}

例如:

```
class A
{
public:
    A(int i, int j)
    {
        A1=i;
        A2=j;
    }
    void print()
    {
        cout<<i<<"", "<<j<<endl;
    }
private:
    int A1, A2;
};

class B
{
public:
    B(int i, int j, int k):a(i, j), b(k){}           //使用成员初始化表
    void print();
private:
    A a;                                           //对象成员
    int b;
};
void B::print()
{
    a.print();
    cout<<b<<endl;
}

void main()
{
    B b(6, 7, 8);
    b.print();
}
```

该程序的输出结果为:

```
6,7
8
```

在这里可以看到 B 类的构造函数完全使用了初始化列表, 初始化列表对对象成员和内置数据类型成员都是可以起作用的。B 类的构造函数在其初始化表里以参数 j 和 j 调用了 A

类的构造函数，从而将成员对象 `m_a` 初始化。

类的内置数据成员的初始化可以采用初始化表或函数体内赋值两种方式，对于内置数据类型的数据成员而言，两种初始化方式的效率几乎没有区别，但后者的程序版式似乎更清晰些。如 `C` 类的声明如下：

```
class C
{
public:
    C(int x, int y); // 构造函数
private:
    int m_x, m_y;
    int m_i, m_j;
}
```

则数据成员在构造函数体内被直接赋值的程序为：

```
C::C(int x, int y)
{
    m_x = x;
    m_y = y;
    m_i = 0;
    m_j = 0;
}
```

比数据成员在初始化表中被初始化(其程序如下)看起来更为自然。

```
C::C(int x, int y):m_x(x), m_y(y)
{
    m_i = 0;
    m_j = 0;
}
```

除了被用来初始化对象成员以外，对象初始化表还有如下的一些使用方法和规则。

- 如果类存在继承关系，派生类必须在其初始化表里调用基类的构造函数。例如：

```
class A
{...
A(int x);                // A 的构造函数
};
class B : public A
{...
B(int x, int y);          // B 的构造函数
};
B::B(int x, int y):A(x)    // 在初始化表里调用 A 的构造函数
{
```



```
...
}
```

- 类的 `const` 常量和引用只能在初始化表里被初始化，因为它不能在函数体内用赋值的方式来初始化。当构造函数体开始执行时，所有 `const` 和引用的初始化必须都已经发生，因此只有将它们放在成员初始化表中指定，才有可能正确地初始化它们。
- 成员对象初始化的次序完全不受它们在初始化表中次序的影响，只由成员对象在类中声明的次序决定。这是因为类的声明是唯一的，而类的构造函数可以有多个，因此会有多个不同次序的初始化表。如果成员对象按照初始化表的次序进行构造，这将导致析构函数无法得到唯一的逆序。

11.8 析构函数

11.8.1 为何需要析构函数

作为一个 C 程序员，可能经常想到初始化的重要性，但很少想到清除的重要性。毕竟，清除一个整型变量时需要做什么？答案是什么也不用做，只需要忘记它。

然而，在面向对象的应用程序中，对于一个曾经用过的对象，仅仅“忘记它”是不安全的。在程序的运行中，它可能修改了某些硬件参数，或在堆中分配了一些内存，如果只是“忘记它”，程序中所做的修改和生成的对象就永远不会消失，始终占据着内存空间。例如 11.6.2 节中的栈，由于调用构造函数而分配的内存并没有进行释放，因此将一直占用，直到程序退出，这是不安全的隐患，因此必须想办法让系统能够在对象不再使用该内存时，回收该内存。

对象是经过构造函数所产生的，在调用了构造函数之后，该对象就会在计算机的内存中占有一席之地。如果对象中申请了动态的内存或占用了其他资源，却没有将它清除，它就会一直存在直到程序结束为止。如果程序中使用大量的对象，却没有适时地将对象从内存中清除，会影响系统与程序的执行效率，因此，将不再使用的对象予以清除是十分必要的。

虽然在 C++ 中清除和初始化同样重要，但是 C++ 语言并没有自动清除内存的机制，必须依靠设计师自行处理，而执行这个任务的就是“析构函数(Destructor Function)”。

前面所介绍的“构造函数”的用途是在内存中创建并初始化一个对象，而现在所要介绍的“析构函数”，则是将已经存在的对象从内存中清除。如果在类中自行定义析构函数，

就能随时利用析构函数清除内存中的对象，让程序员可以自行管理内存。

像构造函数一样，析构函数总是存在的。如果你没有建立自己的析构函数，编译器会自动生成一个。那么为什么还要自定义析构函数呢？答案和为什么需要自定义构造函数一样非常简单：编译器提供的析构函数和构造函数一样，实际上什么也不做。

11.8.2 析构函数的定义

析构函数的命名与构造函数一样，用类的名字作函数名，在前面加上一个“~”，以和构造函数区别。另外，析构函数不带任何参数，也没有返回值，因为析构一个对象不需任何参数，仅仅将它所占有的资源加以释放就是了。下面是一个析构函数的声明。

```
class Y
{
    ...;
public:
    ~ Y() ;//析构函数声明
};
```

当对象超出它的定义范围时，编译器会自动调用析构函数。可以看到，在对象的定义点处构造函数被调用，析构函数被调用的唯一根据则是包含该对象的右括号。因此，应该注意 goto 语句的使用，即使用 goto 语句跳出这一程序块有可能将不会引发析构函数的调用。(是否会出现这种情况取决于编译器的实现，意味着 goto 代码是不可移植的)。下例利用析构函数修补 11.6.2 节中栈的程序，使得内存存在不需要的时候能够自动释放。

```
class Stack
{
public:
    Stack();
    ~Stack();

protected:
    int *Data;
    int Pos;
    int Size;
};

Stack::Stack()
{
    Data=new int[200];
    Pos=0;
    Size=200;
```



```
}  
Stack::~Stack()  
{  
    delete []Data;  
}
```

11.8.3 何时需要使用析构函数

一个需要考虑的问题是何时需要析构函数，一般来讲，可以遵循以下的原则。

- 如果类分配了系统资源而没有相应的自动释放机制，需要进行手工资源释放，那么需要一个析构函数。注意如果有了自动释放机制，例如标准库提供的智能指针“auto_ptr”所管理的内存，那么就不需要析构函数。
- 如果该类必须或者可以作为基类，那么需要一个虚拟析构函数，即便它什么也不做。这是为了确保删除基类指针时析构派生类对象。

除此之外，没理由建立自己的析构函数。

11.9 堆栈和内存分配

11.9.1 内存管理概述

下面总结出了 C++ 程序主要的内存区域。注意，有些名称，比如“堆”(heap)，可能与 C++ 标准中的叫法不一样。内存分成 5 个区，它们分别是代码区、栈区、堆区和全局/静态数据区。

1. 代码区

代码区(code area)存储程序的全部代码，以及字符串常量等在编译期间就能确定的值。在程序的整个生存周期内，常量数据区域中的数据都是可用的。

这个区域内所有的数据都是只读的，任何企图修改本区域数据的行为都会造成无法预料的后果。之所以如此，是因为在实际的实现当中，最底层内部存储格式的实现会使用特定的优化方案。例如，编译器可能把字符串常量只存储一次，而在几个重叠的对象里面引用它。

2. 栈区

栈区(stack area)就是那些存放由编译器在需要的时候自动分配，在不需要的时候自动清

除的变量的存储区，里面的变量通常是局部变量、函数参数等。在函数调用执行过程中产生的局部变量，参数、返回值传递过程中的临时变量等也放在这里，因此栈区也可以看作是函数执行的空间。

这些变量也称作自动存储类型变量(Automatic Variables)或自动变量(对象)。自动变量的存储分配发生在定义它的函数被调用时。分配给自动变量的存储区来自于程序的运行栈，是函数活动记录的一部分。自动对象也被称为具有自动存储持续时间或自动范围的变量。编译器不负责初始化自动变量，未初始化的自动变量包含一个随机的、该存储区上次被使用后的值。在函数结束时，自动变量的活动记录被从运行栈中弹出，与该变量相关联的存储区被真正释放。因此自动变量(对象)的生命期在函数结束时结束，它包含的任何值都被抛弃。

因为与自动对象相关联的存储区在函数结束时被释放，所以应该小心使用自动对象的地址。自动对象的地址不应该被用作函数的返回值，因为函数一旦结束了，该地址就指向一个无效的存储区。同样，当一个自动变量的地址被存储在一个生命期长于它的指针时，在自动变量被释放后，该指针就成为了一个“空悬指针(Dangling Pointer)”。这是一个严重的程序错误，因为空悬指针所指的内容是不可预测的。如果该地址的值正好合适，那么程序不会产生错误而被执行，但是给出的却是绝不可能正确的结果。

一般来说，栈区的分配操作要比动态存储区(比如堆区或者自由存储区)快得多，这是因为栈区的分配只涉及一个指针的递增，而动态存储区的分配涉及较为复杂的管理机制。栈区中，内存一旦被分配，对象就立即被构造好了；对象一旦被销毁，分配的内存也立即被收回。

3. 堆区

堆区(heap area)是一个动态存储区域，使用库函数 `malloc()`和 `free()`和操作符 `new` 和 `delete` 以及一些相关变量来进行分配和回收。在堆区中，对象的生存周期可以比存放它的内存区的生存周期短；这也就是说，程序可以获得一片内存区而不用马上对其进行初始化；同时，在对象被销毁之后，也不用马上收回其占用的内存区。在对象被销毁而其占用的内存区还未被收回的这段时间内，用户可以通过 `void*`型的指针访问这片区域，但是其原始对象的非静态成员以及成员函数都不能被访问或者操纵，因为对象已经不存在了。

操作符 `new` 和 `delete` 是 C++新加入的操作符，它的作用也是用于动态地分配和回收内存空间，但是它与函数 `malloc()`和 `free()`是完全不同的。首先，前者是作为库函数实现的，而后者是操作符，与“+”同一地位；其次，在它们的内部机制和作用上也有着不同。11.9.3

节将详细讲解 new 和 delete 操作符。

要注意的是，虽然在特定的编译器里默认的全局运算符 new 和 delete 也许会按照函数 malloc()和 free()的方式来被实现，但是它们是不同的：用操作符 new 分配的内存不可能由函数 free()安全地回收，反之亦然。

4. 全局/静态数据区

全局的或静态变量和对象所占用的内存区域即全局/静态数据区(global/static area)，它在程序启动的时候才被分配，而且可能直到程序开始执行的时候才被初始化。比如，函数中的静态变量就是在程序第一次执行到定义该变量的代码时才被初始化的。对那些跨越了翻译单元(Translation Unit)的全局变量进行初始化操作的顺序是没有被明确定义的，因而需要特别注意管理全局对象(包括静态类对象)之间的依赖关系。

最后，和前面讲的一样，全局/静态数据区中没有被初始化的对象存储区域可以通过 void*指针来被访问和操纵，但是只要是在对象真正的生存周期之外，非静态成员和成员函数是无法被使用或者引用的。

11.9.2 变量与对象的空间分配时机与初始化

变量与对象的内存分配方式有三种，其空间分配时机及初始化内容如下。

1. 从静态存储区域分配

内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量和 static 变量。

2. 在栈上创建

在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置在处理器的指令集里，效率很高，但是分配的内存容量有限。

3. 从堆上分配

从堆上分配亦称动态内存分配。程序在运行的时候用函数 malloc()或操作符 new 申请任意多少的内存，程序员自己负责在何时用 free()或 delete 释放内存。动态内存的生存期由用户决定，使用非常灵活，但问题也最多。

对于以上三种分配方式，用户要注意内存生命期的问题。

静态分配的区域其生命期是整个软件运行期，就是说从软件运行开始到软件终止退出。只有软件终止运行后，这块内存才会被系统回收。在栈中分配的空间其生命期与这个变量所在的函数和类相关。如果是函数中定义的局部变量，那么它的生命期就是函数被调用时，如果函数运行结束，那么这块内存就会被回收；如果是类中的成员变量，则它的生命期与类实例的生命期相同。在堆上分配的内存，生命期是从调用 `new` 或者 `malloc()` 开始，到调用 `delete` 或者 `free()` 结束。如果不调用 `delete` 或者 `free()`，则这块空间必须到软件运行结束后才能被系统回收，这就造成了内存泄露。

11.9.3 为什么使用 new/delete 操作符

操作符 `new` 和 `delete` 究竟做了些什么？在理解这个问题之前，先来看下面的这段程序。有这么一个程序段：

```
class A
{
public:
    A() {cout<<"A is here!"<<endl;}
    ~A(){cout<<"A is dead!"<<endl;}
private:
    int i;
};

A* pA=new A;
delete pA;
```

在这个简单的程序段里面，操作符 `new/delete` 究竟做了些什么？“`new A`”这句语句实际上做了两件事：首先调用 `new` 操作符，在堆上分配了一个 A 类型对象大小的内存空间；然后调用类的构造函数 `A()`，在这块内存空间上添砖砌瓦，建造起一个 A 类的对象。

对于 `delete` 操作符，它则做了相反的两件事：先调用析构函数 `~A()`，销毁对象；再调用 `delete` 操作符，释放内存。不过需要注意的是，`new` 分配一块内存的时候，并没有对这块内存空间做清零等任何动作，只是拿了过来，这块内存上放的仍然是原来的数据；`delete` 释放内存的时候，也只是将这块内存的使用权归还给操作系统，原来的数据还在其中。所以 `delete` 释放指针 `pA` 之后，`pA` 的值没变，它指向的那块内存的值也没有变。

从 C 程序员转换过来的 C++ 程序员总是有个困惑：`new/delete` 操作符和 C 语言里面的 `malloc()/free()` 函数比起来究竟有什么优势？

其实通过上面的分析，可以看出：

首先，操作符 `new/delete` 实际上做了很多 `malloc()/free()` 函数没有做的事情。

`malloc()/free()`函数只是对内存进行分配和释放;`new/delete` 还负责完成了创建和销毁对象的任务。从这里可以看到, `new` 的工作实际上就是保证原来相互分离的存储分配和初始化工作能够很好地在一起工作。

其次, `new/delete` 操作符的安全性要高于 `malloc()/free()`函数, 因为它返回的就是一个所创建的对象指针, 对于 `malloc()`函数来说返回的则是 `void*`指针。在程序使用它时, 必须要进行强制类型转换, 显然这是一个危险的漏洞。

最后, 用户可以对 `new/delete` 重载, 使内存分配按照指定的方式进行, 这样更具有灵活性, `malloc()`函数则不行。

不过, `new/delete` 也并不是十分完美, 它最大的缺点就是效率低(针对的是默认的分配器), 原因不只是因为它在自由存储区上分配(和栈上对比), 还有以下两个原因。

- `new` 只是对于堆分配器(`malloc/realloc/free`)的一个浅层包装, 没有针对小型的内存分配做优化。
- 默认分配器具有通用性, 它管理的是一块内存池, 这样的管理往往需要消耗一些额外空间。

11.10 拷贝构造函数

类的成员, 可以是任何类型的变量或对象, 因此成员也可以是指针。当类的成员是指针, 指针指向的内存是动态分配的内存, 并且类的对象应用形式出现两个对象间直接赋值的时候, 会出现不同的对象成员指针指向相同内存的错误, 此时只能利用拷贝构造函数来解决问题。

11.10.1 程序出错的原因分析

想想看, 真实世界中的物体可以复制的, 那么 C++ 中的对象呢?

如果在程序中需要多个完全相同的对象时, 该如何处理呢? 一个个慢慢地分别创建? 这可是相当浪费时间且无趣的工作。C++ 语言提供了一种机制, 以允许利用一个对象, 快速地创建多个完全相同的对象, 而且语法相当简单:

类名称 新对象名=旧对象;

这就是 C++ 复制对象的语法, 只要一个简单的“=”就能像普通数据的赋值一样轻易地复制对象, 这个原理如同结构体变量之间的赋值。

例如:

```
int i=10;
int j=i;
```

类似的:

```
class Stack{...//类定义};
A a(i, j);           //使用参数构造对象
A b=a;               //使用已有对象构造新对象
```

与创建一个新的对象一样,复制对象时必须指定对象属于哪一个类,但是不需要任何参数,因为所有的类成员都将与被复制的对象相同。在等号右边的对象就是作为复制来源的对象,复制出来的对象将会与来源对象目前的状态完全相同。

在此之前,类的应用方式都是在程序中直接定义局部或全局的对象进行使用,程序一直运行良好,但是程序并不永远是正确的。看下面的例子。

在本例中,仍然沿用前几节的栈类来说明问题,不同的是,在本例中,栈类对象作为函数的参数出现,代码如下。

```
class Stack
{
    protected:
        int *data;
        int pos;
    public:
        Stack()
        {
            data = new int[200];
            pos = 0;
        }
        ~Stack()
        {
            delete []data;
        }
        void Push(int t)
        {
            data[pos++] = t;
        }
        int Pop()
        {
            return data[--pos];
        }
};
```

数字解忧
PDG


```

void Test(Stack a)
{
}
void main()
{
    Stack x;
    Test(x);
}

```

运行程序，会出现内存不可读的错误，这是为什么呢，程序在哪里出了错误呢？

仔细分析程序就可以发现错误的原因，回想两个结构体变量之间的赋值，在 C 语言中，只能依次对每个结构体成员进行独立的赋值，而在 C++ 中，允许直接利用 “=” 将一个结构体变量的所有成员值直接赋值给另外一个结构体变量(两个变量类型相同)。即：

$$A = B$$

在这个赋值表达式中，结构体 B 的每个成员值都赋值给了结构体 A 的对应成员，类和结构体在底层编译的时候，是做相同处理的，因此对于类对象之间的赋值动作，和结构体变量的赋值动作，没有本质的区别。在上例中，主函数中的对象 x 在调用函数 Test() 时被作为实参传递给函数 Test() 中的形参 a，由于函数的参数是个普通的对象(不是指针)，因此计算机执行的动作可以理解为 $a=x$ ，此时，a 中的每个成员的值都和 x 中对应的成员值是一样的(包括指针成员 data)。当 a 的生命周期结束时，会调用 a 的析构函数，将 a 的成员 data 指向的空间释放给操作系统，不幸的是，该空间其实也是 x 的成员 data 所指向的空间(如图 11-1 所示)，这样，当 x 生命周期结束的时候，会调用 x 的析构函数，而此时，x 的 data 所指向的空间已经释放过了，所以无法再次释放，这就造成了程序出现前述错误。

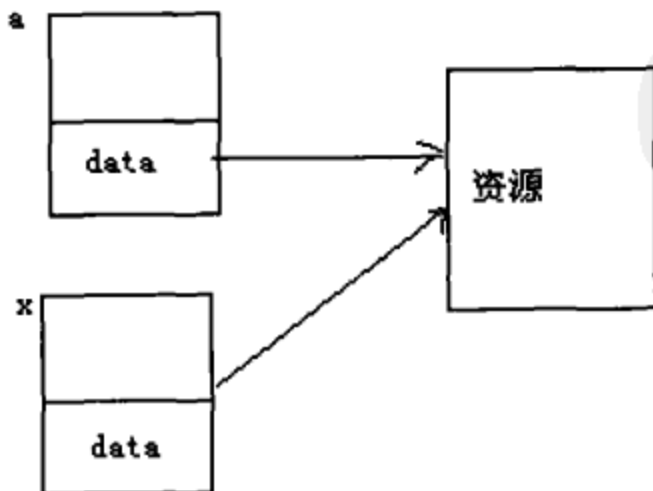


图 11-1 两个对象使用了同一块动态分配的内存

那么，如何来解决这个问题呢？可以这样来考虑，如果在对象赋值的时候，保证新对

象只使用自己的内存，不去占用原对象的内存(仅仅把数据复制过来)，问题就可以解决了。C++恰好提供了一种机制，允许用户控制对象的赋值过程，那就是拷贝构造函数。

11.10.2 拷贝构造函数

在构造函数一节(11.6 节)中曾经讲过，每个类在声明的时候，即使用户不创建构造函数，系统仍默认创建一个无参的构造函数，除非用户覆盖构造函数。如同类的构造函数一样，系统为每个类也创建了默认的拷贝构造函数，原型如下：

```
classname(classname &t);
```

每当对象间进行赋值时，该函数就被自动调用，只是默认情况下，该函数仅仅是对当前对象的每个成员进行简单的赋值为 t 的对应成员值。如果用户覆盖该函数，则对象赋值时将使用新的拷贝构造函数进行工作。

如果将拷贝构造函数改为如下形式：

```
class Stack
{
protected:
    int *data;
    int pos;
public:
    Stack()
    {
        cout <<"构造新对象 " <<endl;
        data = new int[200];
        pos = 0;
    }
    Stack(Stack& s) //拷贝构造函数
    {
        cout<<"拷贝构造"<<endl;
        int i;
        for(i = 0; i<pos; i++)
        {
            data[i] = s.data[i];
        }
        pos = s.pos;
    }
    ~Stack()
    {
        cout <<"析构 "
```



```

        delete []data;
    }
    void Push(int t)
    {
        data[pos++] = t;
    }
    int Pop()
    {
        return data[--pos];
    }
};

void Test(Stack a)
{
    cout <<"在函数内部 Test()\n";
}

void main()
{
    Stack x;
    cout <<"调用 Test()\n";
    Test(x);
    cout <<"从函数返回 Test()\n";
}

```

再运行程序，就没有错误了，运行的结果为：

```

构造新对象
调用 Test()
拷贝构造
在函数内部 Test()
析构
从函数返回 Test()
析构

```

x 对象的创建调用了普通的构造函数，产生了第一行信息。随后便输出第二行信息；主函数 main()调用 Test(x)函数时，发生了从实参 x 到形参 a 的拷贝构造，于是调用拷贝构造函数而得到第三行信息；随后就进入到 Test()的函数体中，产生了第四行信息；从 Test()函数返回时，形参 a 被析构，所以产生了第五行信息；回到主函数后，输出第六行信息；最后主函数结束时，x 对象被析构，所以产生了第七行信息。

通常拷贝构造函数严格限制在只制作拷贝，但是，本程序为了要帮助大家了解其真相，在一些函数体中设置了输出语句。

11.10.3 默认拷贝构造函数

类定义中, 如果未提供自己的拷贝构造函数, 则 C++ 就会提供一个默认拷贝构造函数, 就像没有提供构造函数时, C++ 提供默认构造函数一样。

C++ 提供的默认拷贝构造函数的工作是完成一个成员一个成员的拷贝。默认的拷贝函数所做的工作如下:

```
class Stack
{
protected:
    int *data;
    int pos;
public:
    Stack()
    {
        data = new int[200];
        pos = 0;
    }
    Stack(Stack& s) //默认的拷贝构造函数
    {
        Data = s.data
        pos = s.pos;
    }
    ~Stack()
    {
        delete []data;
    }
    void Push(int t)
    {
        data[pos++] = t;
    }
    int Pop()
    {
        return data[--pos];
    }
};
```

11.10.4 浅拷贝与深拷贝

在默认拷贝构造函数中, 拷贝的策略是逐个成员依次拷贝。但是, 一个类可能会拥有

资源，当其构造函数分配了一个资源(例如堆内存)的时候，就得面临 11.10.1 节中麻烦的局面：两个对象都拥有同一个资源。当对象析构时，该资源将经历两次资源返还。

对于没有覆盖默认拷贝构造函数的 Stack 类，如果有以下应用：

```
void Test(Stack a)
{
    cout <<"在函数内部 Test()\n";
}
void main()
{
    Stack x;
    cout <<"调用 Test()\n";
    Test(x);
    cout <<"从函数返回 Test()\n";
}
```

程序开始运行时，创建 x 对象，x 对象的构造函数从堆中分配空间并赋给数据成员 data；当执行“a=x;”时，因为没有定义拷贝构造函数，于是就调用默认拷贝构造函数，使得 a 与 x 完全一样，并没有新分配堆空间给 a，见图 11-2；主函数结束时，对象逐个析构，析构 a 时，将堆空间返还给系统，析构 x 时，因为这时 x 的 data 指针指向的是系统空间，因此释放时系统报错。

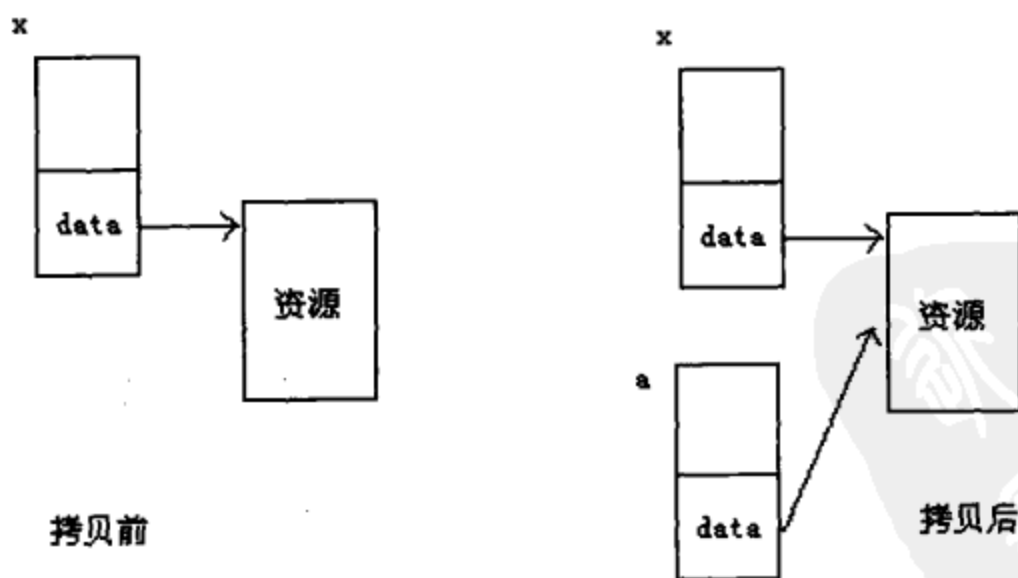


图 11-2 p1/p2 的浅拷贝

创建 p2 时，对象 x 被复制给了 a，但资源并未复制，因此，x 和 a 的成员指针指向同一个资源，这称为浅拷贝。

当一个对象创建时分配了资源，这时就需要定义自己的拷贝构造函数，使之不但拷贝

成员，也拷贝资源。

例如拷贝构造函数：

```
Stack(Stack& s) //拷贝构造函数
{
    cout<<"拷贝构造"<<endl;
    int i;
    for(i = 0; i<pos; i++)
    {
        data[i] = s.data[i];
    }
    pos = s.pos;
}
```

程序开始运行时，创建 x 对象，当用 x 去创建 a 对象时，调用的是自己定义的拷贝构造函数；拷贝构造函数中，复制了资源(堆内存空间)，见图 11-3；当主函数退出时，先后析构 a 和 x，但这时候对象们有其各自的资源，所以，析构函数工作得很好。

创建 a 时，对象 x 被复制给了 a，同时资源也作了复制，因此，x 和 a 的成员 data 指向不同的资源，这称为深拷贝。

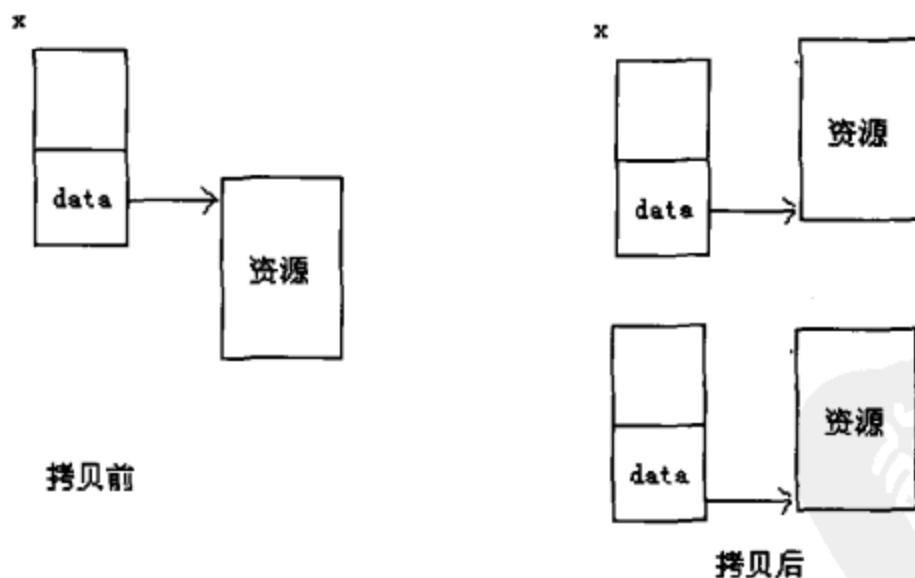


图 11-3 p1/p2 的深拷贝

堆内存并不是唯一需要拷贝构造函数的资源，但它是最常用的一个。打开文件，占有硬设备(例如打印机)服务等也需要深拷贝，它们也是析构函数必须返还的资源类型。因此，一个很好的经验是：如果类需要析构函数来析构资源，则它也需要一个拷贝构造函数。因为通常对象是自动被析构的。如果需要一个自定义的析构函数，那就意味着有额外资源要在对象被析构之前释放。此时，对象的拷贝就不是浅拷贝了。

11.11 临时对象和无名对象

11.11.1 临时对象

当函数返回一个对象时，要创建一个临时对象以存放返回的对象。例如，下面的代码中，返回的 `ms` 对象将产生一个临时对象。

```
Student fn()
{
    //...
    Student ms("a");
    return ms;
}
void main()
{
    Student s;
    s=fn();
    //...
}
```

在这里，系统调用拷贝构造函数将 `ms` 对象拷贝到新创建的临时对象中，见图 11-4。



图 11-4 返回对象的函数运行结束时，创建临时对象存放返回的对象

一般规定，创建的临时对象，在整个创建它们的外部表达式范围内有效，否则无效。也就是说，“`s=fn();`”这个外部表达式，当函数 `fn()` 返回时产生的临时对象拷贝给 `s` 后，临时对象就析构了。例如，下面的代码中，引用 `ms` 不再有效。

```
void main()
{
    Student& refs=fn();
    //...
}
```

因为外部表达式“`Student& refs=fn();`”到分号处结束，以后从函数 `fn()` 返回的临时对象便不再有效，这就意味着引用 `refs` 的实体已不存在，所以接下去的任何对 `refs` 的引用都

是错的。又例如，下面的代码中，一切临时对象都在一个外部表达式中结束。

```
Student fn1();
int fn2(Student&);
void main()
{
    int x;
    x=3*fn2(fn1())+10;
    //...
}
```

函数 `fn1()` 返回时，创建临时对象作为函数 `fn2()` 的实参，此时，在函数 `fn2()` 中一直有效；当函数 `fn2()` 返回一个 `int` 值参与计算表达式时，那个临时对象仍有效；一旦计算完成，赋值给 `x` 后，则临时对象被析构。

11.11.2 无名对象

可以直接调用构造函数产生无名对象。例如，下面的代码在函数 `fn()` 中，创建了一个无名对象。

```
class Student
{
public:
    Student(char*);
    //...
};
void fn()
{
    Student("a"); //此处为无名对象
    //...
}
```

无名对象可以作为实参传递给函数，可以用来拷贝构造一个新对象，也可以初始化一个引用的声明。例如，下面的代码表达了无名对象典型的三种用法。

```
void fn(Student& s);
void main()
{
    Student& refs=Student("a"); //初始化引用
    Student s=Student("b");      //初始化对象定义
    fn(Student("c"));            //函数参数
}
```

主函数开始运行时，第一个执行的是拿无名对象初始化一个引用。由于是在函数内部，所以无名对象作为局部对象产生在栈空间中，从作用域上看，该引用与无名对象是相同的，

它完全等价于 “Student refs=“a”；” 所以这种使用是多余的。

第二个执行的是用无名对象拷贝构造一个对象 s。按理说，C++先调用构造函数 “Student(char);” 创建一个无名对象，然后再调用一个拷贝构造函数 “Student(Student&);” (或许是默认的)创建对象 s；但是，由于是用无名对象去拷贝构造一个对象，拷贝完后，无名对象就失去了任何作用，对于这种情况，C++特别地将其看作为与 “Student s=“b”；” 效果一样，而且可以省略创建无名对象这一步。

第三个执行的是无名对象作为实参传递给形参 s，C++先调用构造函数创建一个无名对象，然后将该无名对象初始化给了引用形参 s 对象，由于实参是在主函数中，所以无名对象是在主函数的栈区中创建，函数 fn()的形参 s 引用的是主函数栈空间中的一个对象。它等价于：

```
Student s("c");
fn(s);
```

如果对象 s 仅仅是为了充当函数 fn()实参的需要，完全可以用第三个执行来代替。当运行到主函数结束的时候，将有一个主函数中的 s 对象和三个无名对象被析构。

11.12 const 成员

在定义一个常量时，const 比 #define 更加灵活。用 const 定义的常量含有数据类型，该常量可以参与逻辑运算。例如：

```
const int a = 100;           // a 是 int 类型
const float MAX=100;        // MAX 是 float 类型
#define b 100                // b 无类型
#define MIN 100              // MIN 无类型
```

除了能定义常量外，const 还有两个“保护”功能。

1. 强制保护函数的参数值不发生变化

以下程序中，函数 f 不会改变输入参数 name 的值，但是函数 g 和 h 都有可能改变 name 的值。

```
void f(String s); //传递值
void g(String &s); //传递引用
void h(String *s); //传递指针
main()
{
```

```

    String name="Dog";
    f(name);           // name 的值不会改变
    g(name);           // name 的值可能改变
    h(name);           // name 的值可能改变
}

```

对于一个函数而言, 如果其'&'或'*'类型的参数只作输入用, 不作输出用, 那么应当在该参数前加上 `const`, 以确保函数的代码不会改变该参数的值(如果改变了该参数的值, 编译器会出现错误警告)。因此上述程序中的函数 `g` 和函数 `h` 应该定义成:

```

void g(const String &s);
void h(const String *s);

```

2. 强制保护类的成员函数不改变任何数据成员的值

以下程序中, 类 `stack` 的成员函数 `Count` 仅用于计数, 为了确保 `Count` 不改变类中的任何数据成员的值, 应将函数 `Count` 定义成 `const` 类型。

```

class Stack
{
public:
    void push(int elem);
    void pop(void);
    int Count(void) const; // const 类型的函数
private:
    int num;
    int data[100];
};
int Stack::Count(void) const
{
    ++ num; // 编译错误, num 值发生变化
    pop();  // 编译错误, pop 将改变成员变量的值
    return num;
}

```

本章小结

本章是 C++ 的一个重点, 主要讲述了类的定义、使用、面向对象编程的概念、数据保护原则和手段、构造及析构函数和拷贝构造函数, 在讲解拷贝构造函数时, 重点分析了栈与堆内存的使用原则, 最后讲述了 `const`。

习 题

1. 设计一个类，要求该类能完成栈的功能，支持数据存储与读取，栈内数据的显示。
2. 修改习题 1 的类，使该类支持任意多(最多受内存的影响)的数据存储。
3. 利用 2 做好的栈，实现整数运算四则表达式的中缀表达式变后缀表达式。
4. 设计一个游戏类，该类能够在屏幕上显示一方块字符，同时可以通过 A、S、W、D 四个按键控制该方块在屏幕上移动。



第 12 章 静态成员与友元

本章内容:

- 静态成员的作用, 以及静态成员变量、静态成员函数的定义与应用。
- 友元的作用, 以及友元函数、友元类的定义与应用。

重点:

- 静态成员的特性。
- 友元的作用。

目的:

利用静态成员, 在各对象之间进行通信; 利用友元, 使外部有条件地访问类的内部, 同时为高级运算符重载奠定语法基础。

12.1 静态成员

12.1.1 为何需要静态成员

有时候, 在程序设计中, 某个特殊类类型所有对象可能有这样的要求: 访问一个全局对象。比如, 可能是要计数在程序的任意一点总共创建了多少个此类类型的对象, 如在即时战略游戏中, 每个兵种单位是一个对象, 要计算当前人口数即是对象数目。又或者类的所有对象拥有一个相同的属性, 如一个指向该类型错误处理代码的入口指针成员, 自然所有同一类对象的这个值是相同的。

在这些情况下, 提供一个所有对象共同使用的全局对象, 比每个类对象都拥有一个独立的数据成员要更为有效, 并且也更加节省了对象存储空间。不过, 虽然这个对象是一个全局对象, 但是它存在的目的只是为了支持该类的实现。因此, 这个全局变量或对象作为类的一部分来实现是更为合理的。

在这种情况下类的静态数据成员提供了一个更好的方案, 就是静态数据成员被当作该类类型的全局对象。对于非静态数据成员, 在每个类对象自己的空间内都有一份拷贝, 而

静态数据成员对每个类类型只有一份拷贝。同全局对象相比，使用静态数据成员有以下两个优势。

- 静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其他全局名字冲突的可能性。
- 可以实现信息隐藏，静态成员可以是私有成员，而全局对象不可以。

12.1.2 静态成员变量

在类中，静态数据成员是类的所有对象中共享的成员，而不是某个对象的成员。因此，静态数据成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。

使用静态数据成员可以节省内存，因为它是所有对象所公有的，因此，对多个对象来说，静态数据成员只需存储一处，供所有对象共用。只要对静态数据成员的值更新一次，就可以保证所有对象存取的就是更新后的相同的值，这样可以大大简化程序，提高效率。

静态数据成员的使用方法和注意事项如下。

- 静态数据成员在定义或说明时前面加关键字 `static`。例如：

```
class MyClass
{
public:
    MyClass(int a, int b, int c);
    void GetNumber();
    void GetSum();
private:
    int A, B, C;
    static int Sum;    //声明静态成员
};
```

- 静态数据成员是静态存储的，它是静态生存期，必须首先对它进行初始化。静态成员初始化与一般数据成员初始化不同。因为静态数据成员是类的成员，而不是对象的成员，所以初始化在类体外进行，使用作用域运算符来标明它所属类，而前面不加关键字 `static`，以免与一般静态变量或对象相混淆。初始化时不加该成员的访问权限控制符 `private`、`public` 等。静态数据成员初始化的格式如下：

<数据类型><类名>::<静态数据成员名> = <值>

例如：

```
class MyClass
{
```

```

public:
    Myclass(int a, int b, int c);
    void GetNumber();
    void GetSum();
private:
    int A, B, C;
    static int Sum;    //声明静态成员
};
int Myclass::Sum = 0;  //静态成员初始化

```

如果静态数据成员的访问权限为公共的，即可在程序中，按一定格式来引用静态数据成员。引用静态数据成员时，采用如下格式：

<类名>::<静态成员名>

下面举一例子来说明静态数据成员的应用。

```

class Myclass
{
public:
    Myclass(int a, int b, int c);
    void GetNumber();
    void GetSum();
private:
    int A, B, C;
    static int Sum;    //声明静态成员
};
int Myclass::Sum = 0;  //静态成员初始化

Myclass::Myclass(int a, int b, int c)
{
    A = a;
    B = b;
    C = c;
    Sum += A+B+C;
}
void Myclass::GetNumber()
{
    cout<<"Number="<<A<<" "<<B<<" "<<C<<endl;
}
void Myclass::GetSum()
{
    cout<<"Sum="<<Sum<<endl;
}
void main()

```

```

{
    Myclass M(1, 2, 3), N(4, 5, 6);
    M.GetNumber();
    N.GetNumber();
    M.GetSum();
    N.GetSum();
}

```

程序运行结果如下：

```

Number=1,2,3
Number=4,5,6
Sum=21
Sum=21

```

从输出结果可以看到 Sum 的值对对象 M 和 N 都是相等的。这是因为在初始化对象 M 时，将对象 M 的三个 int 型数据成员的值求和后赋给了 Sum，于是 Sum 保存了该值。在初始化对象 N 时，对将对象 N 的三个 int 型数据成员的值求和后又加到 Sum 已有的值上，于是 Sum 将保存最后的值。所以，不论是通过对象 M 还是 N 来引用的值都是一样的，即为 21。

12.1.3 静态成员函数

静态成员函数和静态数据成员一样，它们都属于类的静态成员，而不是对象成员。因此，对静态成员的引用不需要用对象名。

在静态成员函数的实现中不能直接引用类中说明的非静态成员，但可以引用类中说明的静态成员。如果静态成员函数中要引用非静态成员时，可通过对象来引用。下面通过例子来说明这一点。

```

class Myclass
{
public:
    Myclass (int a)
    {
        A=a;
        B+=a;
    }
    static void func1(Myclass m); //静态成员函数
private:
    int A; //非静态数据成员
    static int B; //静态数据成员
};

```



```
int MyClass::B=0;           //静态数据成员初始化
void MyClass::func1(MyClass m)
{
    cout<<"A="<<m.A<<endl;
    cout<<"B="<<B<<endl;
}
void main()
{
    MyClass P(5),Q(10);
    P.MyClass::func1(P);    //调用时不用对象名
    MyClass::func1(Q);
}
```

从中可看出,调用静态成员函数可以使用如下两种格式。

- <对象名>.<静态成员函数名>(<参数表>);
- <类名>::<静态成员函数名>(<参数表>);

12.2 友元

12.2.1 为何需要友元

类具有封装和信息隐藏的特性。非成员函数只可以访问类中的公有成员,只有类的成员函数才能访问类的私有成员,程序中的其他函数是无法访问私有成员的。对象的这种数据封装和数据隐藏使对象和外界以一堵不透明的墙隔开,保证了信息的安全。但是这给软件设计者增加了负担,它要求设计者确保每个类提供足够的方法对所有遇到的情况进行处理(任何事物都有利有弊)。

另外,应该看到在某些情况下,特别是在对某些成员函数多次调用时,由于参数传递、类型检查 and 安全性检查等都需要时间开销,而影响了程序的运行效率。数据隐藏还给两个类共享同一函数或数据带来了困难和额外的开销,这是因为每次访问这些共享内容都需要通过函数调用来完成。如果这种访问不经常发生,这类开销还能接受;如果经常发生,就会变得难以容忍。

但是如果将数据成员都定义为公有的,这又破坏了隐藏的特性。因此,就须寻求一种途径使得类以外的对象或函数能够访问类中的私有成员,为了解决上述问题,提出了一种使用友元的方案。C++中的友元为封装及隐藏这堵不透明的墙开了一个小孔,外界可以通过这个小孔窥视内部的秘密。

只要将外界的某个对象说明为某一个类的友元, 那么这个外界对象就可以访问这个类对象中的私有成员。声明为友元的外界对象既可以是另一个类的成员函数, 也可以是不属于任何类的一般函数, 还可以是整个的一个类, 这样, 此类中的所有成员函数都成为友元函数。

友元的作用在于提高程序的运行效率, 但是, 它破坏了类的封装性和隐藏性, 使得非成员函数可以访问类的私有成员。使用友元会使数据封装性受到削弱, 导致程序可维护性变差, 因此一定要慎重使用。

友元类的声明可以放在类声明中的任何位置, 因为友元不是成员, 不受类的保护机制影响, 所以不管在类的哪个部分声明, 都不影响其使用特性。

12.2.2 友元函数

友元函数是一种定义在类外部的普通函数, 但它需要在类体内进行说明, 为了与该类的成员函数加以区别, 在说明时前面加以关键字 `friend`。友元不是该类成员函数, 但是它可以访问类中的私有成员。

友元声明包含在其私有成员可被作为友元的外界对象访问的类的定义中, 也就是将 `friend` 关键字放在函数名或类名的前面。此声明可放在类的公有部分, 也可放在私有部分, 效果是一样的。例如:

```
class point
{
    int x,y;
public:
    point(int,int);
    friend void print(point &p);
};
void print(point &p)
{
    Cout<<"point : "<< p.x<<" , "<< p.y<<endl;
}
```

在这个例子中, `void print()` 被声明为 `point` 类的友元, 于是它就可以访问 `point` 类的成员来显示结果了。

在该例子中的 `point` 类中声明了一个友元函数 `print()`, 它在声明时前边加 `friend` 关键字, 标识它不是成员函数, 而是友元函数。它的定义方法与普通函数定义一样, 而不同于成员函数的定义, 因为它不需要指出所属的类。但是, 它可以访问 `point` 类对象的私有成员 `x` 和 `y`, 在调用友元函数时, 也是同普通函数的调用一样, 不需要像成员函数那样通过对象

调用。

因为友元函数是非成员函数，所以它无法通过 `this` 指针获得对象，因此必须给友元函数传递一个对象变量作为参数，才能获取对象数据并对其操作。

一个独立的友元函数可以访问多个类的数据，但必须同时为这多个类的友元。友元函数还可以是另一个类的成员函数，这种情况可以用于解决两个类之间数据的共享问题。

12.2.3 友元类

除了前面讲过的友元函数以外，友元还可以是类，即一个类可以作另一个类的友元。当一个类作为另一个类的友元时，这就意味着这个类的所有成员函数都是另一个类的友元函数。友元如果被适当地使用，实际上可以增强封装。当一个类的两部分有不同数量的实例或者不同的生命周期时，经常需要将一个类分割成两部分。在这些情况下，两部分通常需要直接存取彼此的数据(这两部分原来在同一个类中，不必增加直接存取一个数据结构的代码；只要将代码改为两个类就行了)。实现这种情况最安全的途径就是使这两部分成为彼此的友元。

友元类声明的形式如下：

```
friend class <友元类名>; 或 friend <友元类名>;
```

声明了友元类后，友元类中的所有成员函数都成为友元函数。例如：

```
#include<iostream.h>
#include<string.h>
class c2;      //类的前导声明
class c1
{
    int v1;
    int v2;
public:
    c1(int a,int b)
    {
        v1 = a;
        v2 = b;
    }
    void point(c2 &t2);
    ~c1()
    {
    }
};
class c2
{
```



```

    int v1;
    int v2;
    friend c1;           //声明类 c1 为类 c2 的友元类
public:
    c2(int a,int b)
    {
        v1 = a;
        v2 = b;
    }
    ~c2()
    {
    }
};
void c1::point (c2 &t)
{
    cout<<"x 的数据 v1 = "<<v1<<"v2 = "<<v2<<endl;
    cout<<"y 的数据 v1 = "<<v1<<"v2 = "<<v2<<endl;
}
void main()
{
    c1 x(1,2);
    c2 y(3,4);
    x.point(y);
}

```

程序运行结果如下:

```

x 的数据 v1 = 1 v2 = 2
y 的数据 v1 = 3 v2 = 4

```

如果按上面所描述的那样定义和使用友元, 就可以使私有的保持私有。不理解的读者在以上这种情形下想避免使用友元, 要么使用公有的数据, 要么通过公有的 `get()` 和 `set()` 成员函数使两部分可以访问数据。而这么做实际上破坏了封装。只有当在类外(从用户的角度)看待私有数据仍“有意义”时, 为私有数据设置公有的 `get()` 和 `set()` 成员函数才是合理的。在许多情况下, 这些 `get()/set()` 成员函数和公有数据一样差劲: 它们仅仅隐藏了私有数据的名称, 而没有隐藏私有数据本身。

同样, 如果将友元函数当作一种类的公共存取函数的语法不同的变种来使用的话, 友元函数就和破坏封装的成员函数一样会破坏封装。换一种说法, 类的友元不会破坏封装的壁垒: 和类的成员函数一样, 它们就是封装的壁垒。

关于友元, 还有以下两点要注意。

- 友元关系是不能传递的。类 B 是类 A 的友元, 类 C 是类 B 的友元, 类 C 与类

A 之间，除非有特别声明，否则没有任何关系，不能进行数据共享。

- 友元关系是单向的。类 B 是类 A 的友元，类 B 的成员函数可以访问类 A 的私有成员和保护成员，反之，类 A 的成员函数却不可以访问类 B 的私有成员和保护成员。



第 13 章 继承与多态

本章内容:

- 继承与派生的基本概念。
- 派生类的声明和访问权限。
- 派生类构造函数和析构函数的定义及使用。
- 多态的概念, 虚函数的定义及使用。
- 多继承的声明、构造函数和析构函数的定义及使用。
- 虚基类的作用、定义和使用。

重点:

- 继承前后访问属性的变化。
- 多态。

目的:

掌握代码重用的技巧, 同时掌握面向对象的继承与多态这两个特性。

在第 11 章“类与对象”中, 介绍了面向对象的概念, 认识了什么是类及为何要用类取代以往的程序结构。接下来, 将介绍面向对象中另两个重要特性: 继承与派生。

所谓的继承就是指以现有的类为基础来创建新类。继承是一种联结类的层次模型, 并且允许和鼓励类的重用, 它明确表述了事物之间的共性和联系。一个新类是从现有的类中派生, 新类继承了原始类的特性, 这个过程称为类的继承。

通常称被其他类继承的类为基类或父类(Base Class), 而继承自别的类而成的类则称为派生类或子类(Derived Class)。派生类可以从它的基类那里继承基类的所有方法和属性, 并且派生类同时也能依其所需, 修改或增加新的方法使之适合特殊的需要。继承体现了大自然中一般与特殊的关系, 也大大增加了代码的可重用性。比如说, 所有的 Windows 应用程序都有一个窗口, 它们可以看作都是从同一个窗口类派生出来的。但是有的应用程序用于文字处理, 有的应用程序用于绘图, 这些不同的窗口表现出完全不同的外观和特性。这就是由于派生出了不同的子类, 各个子类添加了不同的特性, 这就是多态的现象。

一个派生类也可能不止继承了一个类, 如果同时继承了多个基类, 这种情况称为多重

继承，本章的后半部分将会介绍多重继承。

13.1 继承与派生的概念

在日常生活中当人们面临一笔数目庞大的数据时，如果它是未经分类的，不但本身就占用了极大的空间，同时在工作过程中也极有可能出错。所以进行分类是非常必要的，分类也早已是人类自然思维的重要习惯。好像在生物学上为了管理世界上所有的生物，制定了“界、门、纲、目、科、属、种”7个层次，有相同特性者归于同一类。例如在分类“界”时，凡符合“多细胞真核类，行异养生活”者便归于动物界，但是属于动物界的生物彼此间仍有差异，因此继续向下一层分类。在动物界中的生物有许多符合“左右对称，有许多环节，表面有外骨骼”的条件，因此将它们归为“节肢动物门”。此时许多的蜘蛛、蝴蝶和蜻蜓都归于其下，然而这些动物们间仍存在着极大的差异，分析的结果，发现凡是蜘蛛都拥有“身体分为两部分，脚有四对”的特性，而蝴蝶和蜻蜓们则有“身体有三部分，头部有触角、胸部有脚三对翅两对”的特性。故将它们分成两类，即“昆虫纲”和“蜘蛛纲”。生物的分类如图13-1所示。

由此可见分类的条件是由上层的一般性质到下层的特殊性质，以确定至最下层时能清楚分类出所有种类的生物。同时观察以上的结果会发现，属于昆虫纲的动物及蜘蛛纲的动物会有属于节肢动物门“左右对称，有许多环节，表面有外骨骼”的性质，而且也有动物界“多细胞真核类，行异养生活”的性质。在下层的分类会自动包含它上层的一般性质，同时加上自己的特殊性质，换句话说，在下层的分类“继承”了在它上层分类的特性。

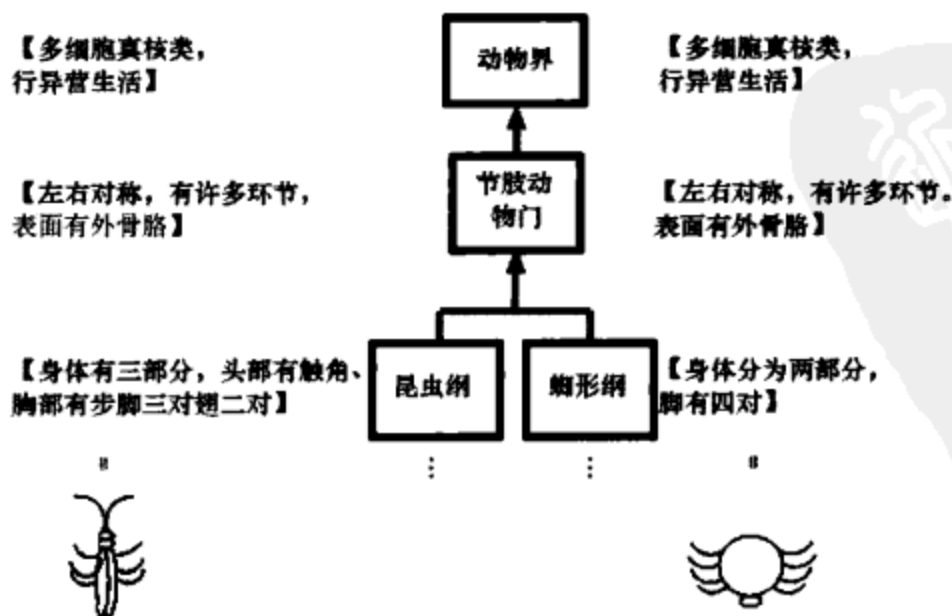


图 13-1 生物的分类

事实上在编写程序时,也可以采取相似的分类方法,把上例中动物的分类想成是 C++ 中的类,把动物的性质想成是类的数据成员和成员函数。将数据中有相同的部分者归为一个类,如果其中仍存在着歧异则继续分类下去,最后形成一个层次式的结构。在上层的类拥有一般性质的成员,下层的类则除了拥有特殊性质的成员外也会包括上层类的一般性质,这种过程就是继承。

利用继承来编写程序时,除了能减少类内数据的重复外,同时也能让程序更容易管理,因为它能有效处理程序的扩充问题。一个程序只有极少机会不会再扩充其数据或功能,当扩充时希望能尽量不要更改到原始的程序代码,以免造成牵一发而动全身的情况。例如新发现了一种千只脚的生物,它有着节肢动物门的一般性质,也不属于任一个现有的物种,所以必须另外归为一个类。在以往面向过程的程序写作时,可能会写成:

```
switch(type)
{
    case "昆虫纲": ...;           //原有的处理代码
    case "蛛目纲": ...;
    ...
    case "千脚纲": ...;           //新增的纲和处理代码
}
```

然而要新增的并不只是 switch 的部分而已,同时也得更更改一些数据类型(例如,原先以字符类型来存储脚的个数,不料这种新生物有一千只脚,于是只得改用整型来存储)或变量的值(例如类型的总个数)等。当人们对这些部分进行更改时,原有的其他程序代码并不知道有哪些地方被更改了,一旦它们仍以原来的方法去访问这些变量时,错误就发生了。为了解决这些错误,又会再更改这些原有的程序代码,却又引发出更多的错误。由此可见,在面向过程的代码中,想要更改程序源代码是非常麻烦和危险的。而在面向对象程序设计中,新的类只需继承原有类即可,如图 13-2 所示。

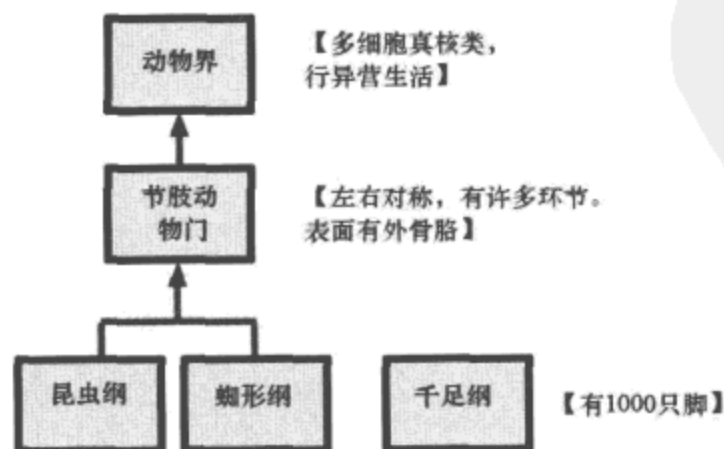


图 13-2 类的继承示意

13.2 继承的实现方式

通过继承机制，可以利用已有的数据类型来定义新的数据类型。所定义新的数据类型不仅拥有新定义的成员，而且还同时拥有旧的成员。通常称已存在的用来派生新类的类为基类，又称为父类。由已存在的类派生出的新类称为派生类，又称为子类。

在 C++语言中，一个派生类可以从一个基类派生，也可以从多个基类派生。从一个基类派生的继承称为单继承；从多个基类派生的继承称为多继承。单继承派生类的定义格式如下：

```
class <派生类名> : <继承方式> <基类名>
{
    ...//派生类新定义成员
};
```

其中，“<派生类名>”是新定义的一个类的名字，这个类是从“<基类名>”指定的基类中派生的，并且按指定的“<继承方式>”派生。继承方式常使用如下三种关键字给予表示。

- **public**：表示公有继承。
- **private**：表示私有继承。
- **protected**：表示保护继承。

例如：如果类 A 是基类，类 B 是 A 的派生类，那么 B 将继承 A 的全部方法和属性。示例程序如下。

```
class A
{
public:
    void Func1(void);
    void Func2(void);
};
class B : public A
{
public:
    void Func3(void);
    void Func4(void);
};
main()
{
    B b;           // B 的一个对象
```




```

b.Func1();      // B 从 A 继承了函数 Func1
b.Func2();      // B 从 A 继承了函数 Func2
b.Func3();      // B 所特有的函数 Func3 和 Func4
b.Func4();
}

```

在派生定义了 B 类后, 就可以使用它来声明对象。通过 B 类的对象不仅可以调用它独有的成员, 还可以使用由 A 类继承来的成员, 而不需再次定义它们。

13.3 继承类的构造与析构

13.3.1 继承类的构造

如 13.2 节例子, class B 由 class A 继承而来, 它的一个对象在内存中所占据的空间如图 13-3 所示: B 是 A 的派生类, 因此拥有 A 的一切成员。从内存的角度上来看, 就是在它的内存空间中拥有一份 A 类对象的完整拷贝, 而在其后才是 B 类特有的成员。

面对继承而来的类, 当创建它的对象时, 疑问产生了: 如何构造派生类的对象, 基类成员部分、派生类成员部分的构造又是分别由谁来负责的?

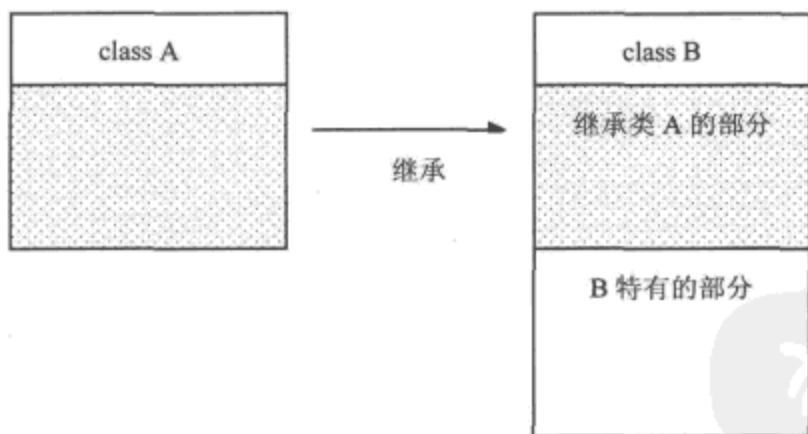


图 13-3 类的继承与派生类的构造示意

回想一下前面构造函数的概念: 构造函数是一个特殊的成员函数, 它在创建对象时自动执行, 进行对象的初始化工作。在派生类的构造函数也是一样的情况, 不过由于它也要同时初始化基类的成员, 这就使得情况变得稍微复杂了些。

由于构造函数是对整个对象进行初始化, 因此当创建派生类的对象时, 必须同时初始化其继承的基类的成员空间才行。作为一般规则, 派生类构造函数应该不能直接向一个基类数据成员赋值, 否则, 派生类的构造函数实现还要考虑基类的结构实现, 将更加难于正确地修改或扩展基类的实现, 这就违背了 C++ 继承机制的本意。

于是立刻可以想到，基类本身也带有初始化的工具——构造函数。初始化基类的成员空间最简单有效的方法就是调用基类的构造函数。当处理无参数的构造函数时，C++编译器会自动执行其基类的构造函数后才执行本身的构造函数；先完成基类部分成员初始化，然后是派生类的部分。然而当构造函数含有一个以上的参数时，编译器便不会先去执行基类含有相同参数的构造函数，此时就得完全依靠手动去调用它。

与构造函数相对的是析构函数，由于析构函数没有参数，所以不会发生以上的参数调用问题。析构函数的调用顺序与构造函数相反：对象析构时，编译器会先执行本身的析构函数，然后才轮到执行基类的析构函数。看下面的示例。

```
class base                                //基类
{
public:
    base(void)
    {
        cout<<"base constructing"<<endl;
    }
    ~base(void)
    {
        cout<<"base destructing"<<endl;
    }
};
class member                             //对象成员类
{
public:
    member(void)
    {
        cout<<"member_class constructing"<<endl;
    }
    ~member(void)
    {
        cout<<"member_class destructing"<<endl;
    }
};
class derived :public base                //派生类
{
    member member_class;                  //派生类拥有的私有对象成员
public:
    derived(void)
    {
        cout<<"derived constructing"<<endl;
    }
    ~derived(void)
    {
```

```
        cout<<"derived destructing"<<endl;
    }
};
void main()
{
    derived d;
}
```

执行以上代码, 结果如下:

```
base constructing
member_class constructing
derived constructing
derived destructing
member_class destructing
base destructing
```

由例子可以看出, 当派生类是由一个或多个(多继承时)基类对象部分以及派生类部分构成时, 在派生类构造时, 构造函数的调用顺序总是如下。

(1) 基类的构造函数。如果有多个基类(多继承), 则构造函数的调用顺序是某类在类派生表中出现的顺序, 而不是它们在成员初始化表中的顺序。

(2) 类对象成员的构造函数。如果有多个成员类对象, 则构造函数的调用顺序是对象在类中被声明的顺序, 而不是它们出现在成员初始化表中的顺序。

(3) 派生类的构造函数。

而析构函数的调用顺序则与之正好相反。

(1) 派生类的析构函数。

(2) 类对象成员的析构函数。

(3) 基类的析构函数。

尤其是当有多个类之间有着复杂的继承关系时, 要特别小心对象及对象成员间的构造析构顺序, 否则极易产生令人意想不到的错误。

13.3.2 构造函数的参数传递

在 13.3.1 节讲到, 作为面向对象的设计思想要求, 派生类应当只需要考虑自身成员的初始化, 而基类的成员则只是简单地通过自动调用基类的构造函数来实现。基类设计者的责任就是提供一组适当的基类构造函数。

但是, 上面的自动调用只是一种内部机制, 默认调用的只能是不带参数的默认构造函数。对于构造函数的任务——初始化类的成员来说, 如何为派生类和基类的成员赋予适当

的初值呢?

第一种方法是默认参数值, 和一个简单类的构造一样, 考虑将其初始化为一个默认值或是用户想要的任何内容。默认内容可以像下面这样。

```
class people          //人类
{
    long ID;          //身份证号
    char name[20];     //姓名
    char sex;          //性别
public:
    people(long num=0, char* n="Carl", char s='m')    //带默认参数的构造函数
    {ID=num; strcpy(name,n), sex=s;}
    void display()
    {
        cout<<"people: ID="<<ID;
        cout<<" name="<<name<<" sex="<<sex<<endl;
    }
};

class student:public people    //学生类, 具有基类 people 类的全部数据结构和操作
{
    int s_ID;                //学号
public:
    int s_ClassID;           //班级

    student(int sID=0, int sc=0) //构造函数
    {
        s_ID=sID;
        s_ClassID=sc;
    }
    void s_display()
    {
        cout<<"student: "<<endl;
        display();
        cout<<" s_ID="<<s_ID<<" class="<<s_ClassID<<endl;
    }
};
```

基类和派生类的构造函数都带有默认的参数, 创建的对象将被初始化为同一个值。运行以下程序。

```
void main()
{
    student p;
    p.s_display();
    student s;
```

```

        s.s_display();
    }

```

可以看到不论是基类成员还是派生类成员，都被很好地赋值了。但是这样构造出的所有的对象都是一个样。如果打算将对象初始化为指定的内容，则派生类的构造函数需要接受所有这些参数，不论是派生类的成员值还是基类的成员值，而后通过某种机制将用于构造基类函数的参数传给基类的构造函数。

这种机制就是前面提到过的成员初始化表。使用成员初始化表来将参数传递给基类的构造函数。于是，上面的例子就可以修改如下：

```

class people                                //人类
{
    long ID;                                //身份证号
    char name[20];                          //姓名
    char sex;                               //性别
public:
    people(long num=0, char* n="Carl", char s='m') //基类构造函数
    { ID=num; strcpy(name, n), sex=s; }
    void display()
    {
        cout<<"people: ID="<<ID;
        cout<<" name="<<name<<" sex="<<sex<<endl;
    }
};
class student:public people                //学生类，继承有基类 people 类的全部属性和方法
{
    int s_ID;                              //学号
public:
    int s_ClassID;                         //班级
    //带成员初始化表的派生类构造函数
    student(long ID, char* name, char s='m', int sID=0, int sc=0):people(ID, name, s)
    {
        s_ID=sID;
        s_ClassID=sc;
    }
}

```

现在，在程序执行中，就可以使用带参数的方式来创建派生类对象了，例如：

```
student first(0001, "Tom", "w", 0002, 01);
```

上面例子中，student 类由 people 类派生而来，基类与派生类的构造函数分别为：

```
people(long num=0, char* n="Carl", char s='m')
```

和

```
student(long ID,char* name,char s='m',int sID=0,int sc=0):people(ID,name,s)
```

这里冒号后面的 `people(ID,name,s)` 语句完成了对基类成员的初始化，它表示学生类构造函数所接受的五个参数中的三个要被用来构造基类的成员，而并没有用于派生类的构造。实际上，这就是一个基类构造函数的调用，产生了一个基类的无名对象，并接着用它来填充了派生类对象中基类部分的空间。

这里派生类的构造函数的初始化列表中列出的均是直接基类的构造函数。也就是说如果 B 类继承 A 类，C 类又继承 B 类，那么它们的构造函数应该如下。

```
class A::A(a)
{
    _a=a;
}

class B::B(a,b):A(a)
{
    _b=b;
}

class C::C(a,b,c):B(a,b)
{
    _c=c;
}
```

每一级派生类只负责自己部分的初始化，而其他都交给基类来完成。

13.4 基类访问控制

派生类有三种继承方式：公有继承(public)、私有继承(private)和保护继承(protected)。

1. 公有继承

公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，它们都保持原有的状态，而基类的私有成员仍然是私有的。

2. 私有继承

私有继承的特点是基类的公有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问。

3. 保护继承

保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成员函数或友元访问，基类的私有成员仍然是私有的。表 13-1 列出了三种不同的继承方式的基类和派生类特性。

表 13-1 不同继承方式的基类和派生类特性

继承方式	基类特性	派生类特性
公有继承	public	public
	protected	protected
	private	不可访问
私有继承	public	private
	protected	private
	private	不可访问
保护继承	public	protected
	protected	protected
	private	不可访问

为了进一步理解三种不同的继承方式在其成员可见性方面的区别，下面从三种不同角度进行讨论。

对于公有继承方式：

- 基类成员对其对象的可见性：公有成员可见，其他不可见。这里保护成员同于私有成员。
- 基类成员对派生类的可见性：公有成员和保护成员可见，而私有成员不可见。这里保护成员同于公有成员。
- 基类成员对派生类对象的可见性：公有成员可见，其他成员不可见。

所以，在公有继承时，派生类的对象可以访问基类中的公有成员；派生类的成员函数可以访问基类中的公有成员和保护成员。这里一定要区分清楚派生类的对象和派生类中的成员函数对基类的访问是不同的。

对于私有继承方式：

- 基类成员对其对象的可见性：公有成员可见，其他成员不可见。
- 基类成员对派生类的可见性：公有成员和保护成员是可见的，而私有成员是不

可见的。

- 基类成员对派生类对象的可见性：所有成员都是不可见的。

所以，在私有继承时，基类的成员只能由直接派生类访问，而无法再往下继承。

对于保护继承方式：

这种继承方式与私有继承方式的情况相同，两者的区别仅在于对派生类的成员而言，对基类成员有不同的可见性。

上述所说的可见性也就是可访问性。关于可访问性还有另的一种说法，这种规则中，称派生类的对象对基类访问为水平访问，称派生类的派生类对基类的访问为垂直访问。一般规则如下。

- 公有继承时，水平访问和垂直访问对基类中的公有成员不受限制。
- 私有继承时，水平访问和垂直访问对基类中的公有成员也不能访问。
- 保护继承时，对于垂直访问同于公有继承，对于水平访问同于私有继承。

对于基类中的私有成员，只能被基类中的成员函数和友元函数所访问，不能被其他的函数访问。

13.5 多态与虚函数

龙生九子，子子不同。这个道理说的是世界上没有两个或两个以上的事物是完全相同的，事物之间有相同相似的一面，当然也有不同、相异的一面。只有将不同的和相同的都考虑进来，才能完整地表达事物。

事物的属性有着不同的一面。人感知事物的一个途径是观察，用对不同光线的反应来识别物体，给出事物的一个属性，如颜色。有时候一些物体的颜色是固定的，比如白色的云，蔚蓝的天空，碧绿的草地等。

但有些方面是不确定的，有一句话说道：“月有阴晴圆缺，人有悲欢离合”，这句话还不足以说明这个道理。中文里有时候会省略一些部分，“今天你吃了没有？”吃的是什么，“早饭”，“下午茶”，“面包”，不知道。这里将吃东西抽象化了，这句话可以在早上说，中午说，晚上说，任何一个适合的时间段都可以说。感兴趣的地方出现了，大多数情况下，人们不知道吃什么东西，什么时候吃，但是却可以使用它，简要地表达一下你的问候。这样简单地设下伏笔，在需要的环境下再表现出具体的行为，这种用法被广泛运用在生活中。“我砍！”表达了一个人要做的动作，具体砍什么，只有感兴趣的人才会去关注；“我生病了。”表达了一个人的状态，具体生的什么病，不是人人都想知道的。

多态就是指事物不同的方面。

封装、继承和多态是 C++ 的三大特性，也是面向对象语言的三大基本特性。多态应当说是一种考虑问题的方法、一种思想，它提供了与具体实现相隔离的另一类接口，即把“what”从“how”中分离开来。多态性提高了代码的组织性和可读性，同时也可使得程序具有可生长性，这个生长性不仅指在项目的最初创建期可以“生长”，而且希望项目具有新的性能时也能“生长”。

封装是通过特性和行为的组合来创建新数据类型的，通过让细节私有来使得接口与具体实现相隔离。这类机构对于有过程程序设计背景的人来说是非常有意义的。而虚函数则根据类型的不同来进行不同的隔离。

第 12 章提到，继承如何允许把对象作为它自己的类型或它的基类类型处理。这个能力很重要，因为它允许很多类型(从同一个基类派生的)被等价地看待，它们就像是一个类型，允许同一段代码同样地工作在所有这些不同类型上。虚函数反映了一个类型与另一个类似类型之间的区别，只要这两个类型都是从同一个基类派生的。这种区别是通过其在基类中调用函数的表现不同来反映的。

多态性在 C++ 中是用虚函数来实现的。在这一节中，将从最基本的内容开始学习虚函数。虚函数加强类型概念，而不是只在结构内和墙后封装代码，所以毫无疑问，对于新 C++ 程序员，它们是最困难的概念。然而，它们也是理解面向对象程序设计的转折点。如果不用虚函数，就等于还不懂得面向对象。

13.5.1 为什么使用虚函数

看下面的例子：

```
class CBase
{
public:
    void who()
    {
        cout<<"this is the base class!\n";
    }
};
class CDerive1 : public CBase
{
public:
    void who()
    {
        cout<<"this is the derive1 class!\n";
    }
}
```



```
};  
class CDerive2 : public CBase  
{  
public:  
    void who( )  
    {  
        cout<<"this is the derive2 class! \n";  
    }  
};  
void main()  
{  
    CBase obj, *p;  
    p = &obj1;  
    p = &obj2;  
    p -> who();  
    p = &obj3;  
    p -> who();  
    obj2.who();  
    obj3.who();  
    return;  
}
```

程序运行结果如下:

```
this is the base class!  
this is the base class!  
this is the base class!  
this is the derive1 class!  
this is the derive2 class!
```

由此例可看出, 通过对象指针进行的普通成员函数调用仅仅与指针的类型有关, 而与此刻正指向什么对象无关。要想实现当指针指向不同对象时执行不同的操作, 就必须将基类中相应的成员函数定义为虚函数。

13.5.2 虚函数

把函数体与函数调用相联系称为绑定(binding)。当绑定在程序运行之前(由编译器和连接器)完成时, 称为早绑定。读者可能没有听到过这个术语, 因为在过程语言中是不会有: C 编译只有一种函数调用, 就是早绑定。

上面程序中的问题是早绑定引起的, 因为编译器在只有基类指针时不知道正确地调用函数。

解决方法被称为晚绑定, 这意味着绑定在运行时发生, 基于指向对象的类型。晚绑定

又称为动态绑定或运行时绑定。当一个语言实现晚绑定时，必须有一种机制在运行时确定对象的类型和合适的调用函数。这就是，编译器还不知道实际的对象类型，但它插入能找到和调用正确函数体的代码。晚绑定机制因语言而异，但可以想象，一些种类的类型信息必须装在对象自身中。

在 C++ 中，为了引起晚绑定，C++ 要求在基类中声明这个函数时使用 `virtual` 关键字，将它定义为虚函数。晚绑定只对 `virtual` 起作用，而且只发生在使用一个基类的地址，并且这个基类中有 `virtual` 函数时，尽管它们也可以在更早的基类中定义。

为了创建一个 `virtual` 成员函数，可以简单地在这个函数声明的前面加上关键字 `virtual`。对于这个函数的定义不要重复，在任何派生类函数重定义中都不要重复它(虽然这样做无害)。

如果一个函数在基类中被声明为 `virtual`，那么在所有的派生类中它都是 `virtual` 的。在派生类中 `virtual` 函数的重定义通常称为覆盖。

虚函数用来表现基类和派生类成员函数之间的一种关系。虚函数的定义在基类中进行，在需要定义为虚函数的成员函数的声明前冠以关键字 `virtual`。

基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、参数类型及参数的先后顺序，都必须与基类中的原型完全相同。

13.5.3 重载、隐藏与覆盖

虚函数是重载的一种表现形式，是一种动态的重载方式。

一般的重载函数，函数的返回类型及所带的参数必须至少有一样不完全相同，只需函数名相同即可。

基类中定义的虚函数在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、参数类型及参数的先后顺序，都必须与基类中的原型完全相同。

重载虚函数时，若与基类中的函数原型出现不同，系统将根据不同情况分别处理。

- 若仅仅返回类型不同，其余相同，系统会当作出错处理。
- 若函数原型不同，仅仅函数名相同，系统会认为是一般的函数重载，将丢失虚特性。

在前面讲过函数重载的概念，这里的覆盖与它类似。比较重载与覆盖两者之间的特征，可得出它们的区别。

重载(overload)的特征如下。

- 相同的范围(在同一个类中)。
- 函数名相同参数列表不同。
- virtual 关键字可有可无。

覆盖(override)是指派生类函数覆盖基类函数, 覆盖的特征如下。

- 不同的范围(分别位于派生类与基类)。
- 函数名和参数都相同。
- 基类函数必须有 virtual 关键字(若没有 virtual 关键字则称之为隐藏 hide)。

如果基类有某个函数的多个重载版本, 若用户在派生类中覆盖了基类中的一个或多个函数版本, 或是在派生类中重新添加了新的函数版本(函数名相同, 参数不同), 则所有基类的重载版本都被屏蔽, 在这里称为隐藏(hide)。所以, 在一般情况下, 用户想在派生类中使用新的函数版本又想使用基类的函数版本时, 应该在派生类中重写基类中的所有重载版本。若是不想重写基类重载的函数版本, 则应该使用显式声明基类名字空间作用域。

以下两个示例显式声明基类名字空间。

其一:

```
#include <iostream>
using namespace std;
class Basic
{
public:
    void fun(){cout << "Base::fun()" << endl;}           //重载
    void fun(int i){cout << "Base::fun(int i)" << endl;} //重载
};
class Derive :public Basic
{
public:
    using Basic::fun;
    void fun(){cout << "Derive::fun()" << endl;}
    void fun2(){cout << "Derive::fun2()" << endl;}
};
int main()
{
    Derive d;
    d.fun();           //正确
    d.fun(1);          //正确
    return 0;
}
```

输出结果:

```
Derive::fun()
Base::fun(int i)
Press any key to continue
```

其二:

```
#include <iostream>
using namespace std;
class Basic
{
public:
    void fun(){cout << "Base::fun()" << endl;}           //重载
    void fun(int i){cout << "Base::fun(int i)" << endl;} //重载
};
class Derive :public Basic
{
public:
    using Basic::fun;
    void fun(int i,int j){cout << "Derive::fun(int i,int j)" << endl;}
    void fun2(){cout << "Derive::fun2()" << endl;}
};
int main()
{
    Derive d;
    d.fun();           //正确
    d.fun(1);          //正确
    d.fun(1,2);        //正确
    return 0;
}
```

输出结果:

```
Base::fun()
Base::fun(int i)
Derive::fun(int i,int j)
Press any key to continue
```

事实上, C++编译器认为, 函数名相同参数不同的函数之间根本没有什么关系, 它们根本就是两个毫不相关的函数, 只是 C++语言为了模拟现实世界, 为了让程序员更直观地思维处理现实世界中的问题, 才引入了重载和覆盖的概念。重载是在相同的名字空间作用域下, 而覆盖则是在不同的名字空间作用域下, 比如基类和派生类即为两个不同的名字空间作用域。在继承过程中, 若发生派生类与基类函数同名问题, 便会发生基类函数的隐藏。当然, 这里讨论的情况是基类函数前面没有 `virtual` 关键字。在有 `virtual` 关键字时的情形另

做讨论。

继承类重写了基类的某一函数版本，以产生拥有自己功能的接口。此时 C++编译器认为，现在既然要使用派生类自己重新改写的接口，那基类的接口就不提供给用户了(当然用户可以用显式声明名字空间作用域的方法)，而不会理会基类的接口是有重载特性的。若是要在派生类里继续保持重载的特性，那就需要自己再给出接口重载的特性。所以在派生类里，只要函数名一样，基类的函数版本就会被无情地屏蔽。在编译器中，屏蔽是通过名字空间作用域实现的。

所以，在派生类中要保持基类的函数重载版本，就应该重写所有基类的重载版本。重载只在当前类中有效，继承会失去函数重载的特性。也就是说，要把基类的重载函数放在继承的派生类里，就必须重写。

“隐藏”是指派生类的函数屏蔽了与其同名的基类函数，具体规则也来做一小结。

- 如果派生类的函数与基类的函数同名，但是参数不同，此时，若基类无 `virtual` 关键字，基类的函数将被隐藏。注意别与重载混淆，虽然函数名相同参数不同应称之为重载，但这里不能理解为重载，因为派生类和基类不在同一名字空间作用域内。这里理解为隐藏。
- 如果派生类的函数与基类的函数同名，但是参数不同，此时，若基类有 `virtual` 关键字，基类的函数将被隐式继承到派生类的虚表中。此时派生类虚表中的函数指向基类版本的函数地址。同时这个新的函数版本添加到派生类中，作为派生类的重载版本。但在基类指针实现多态调用函数方法时，这个新的派生类函数版本将会被隐藏。
- 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有 `virtual` 关键字，此时，基类的函数被隐藏(注意别与覆盖混淆，这里理解为隐藏)。
- 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数有 `virtual` 关键字，此时，基类的函数不会被“隐藏”，而要理解为覆盖。

13.5.4 虚函数的限制

基类中的某个成员函数被声明成虚函数后，此虚函数可以在其派生类中被重载，并可以被重新定义，但是函数原型必须相同(包括函数的返回类型)，否则系统会把其作为一般的函数重载而失去虚特性。例如：

```
#include<iostream.h>
#include<stdio.h>
class A
{ int x,y;
public:
```

```

    virtual void show(int a=0){cout<<"A: "<<endl; }    //声明为虚函数
};
class B:public A
{public:
    virtual void show(){ cout<<"B: "<<endl; } //一般的成员函数重载, 不是虚函数重载
};
class C:public B
{public:
    void show(int b=0){ cout<<"C: "<<endl; }    //虚函数重载
};
class D:public C
{public:
    void show(int b=0){ cout<<"D:\n"; }        //虚函数重载
};
void main()
{
    A a,*pa=&a;
    B b; C c; D d;
    pa->show();
    pa=&b; pa->show(); //执行基类 A 的 show() 函数
    pa=&c; pa->show();
    pa=&d; pa->show();
} 执行结果: A:
        A:
        C:
        D:

```

例如:

```

#include<iostream.h>
class Base
{ public:
    virtual Base* afn()
        { cout<<"This is Base class.\n"; return this; }
};
class SubClass:public Base
{ public:
    Base* afn()
        { cout<<"This is SubClass.\n"; return this; }
};
void test(Base& x)
{
    Base *b;
    b=x.afn();
}
void main()
{
    Base bc;
    SubClass sc;

```

```
    test(bc);  
    test(sc);  
}
```

运行结果如下:

```
This is Base class.  
This is SubClass.
```

其他需要注意的有以下几点:

- 关键字 `virtual` 只能对成员函数、基类中非静态的公有成员函数进行声明。
- 内联函数不能声明成虚函数。
- 构造函数不能声明成虚函数。

13.6 多 继 承

虽然在软件设计中,许多书籍都推荐优先使用组合而不是继承,然而继承仍然拥有许多天然的优势:对基类成员的自动拥有,不用像组合那样要显示地去转向调用所需复用的成员,从而平添更多的代码。

多重继承在某些情况下,可以使程序设计具有更多的灵活性。下面讨论一些多重继承中的问题及解决办法。

假设实现了一个抽象基类 A,然后由此派生出了诸多的实现类,如 A1、A2、A3,在项目的起初,这些 A 的具体类工作很好,软件模块也依赖于这一个抽象基类 A。随着项目的进行,又进入了另一个模块的开发。也许起先考虑的不周,又或者设计师在设计时出现了其他什么,这里又要使用 A1、A2、A3 了。但是这时发现,抽象基类 A 的这些接口方法已经不能满足这一个模块的功能要求了,在这个新的模块中,需要另外一些通用的方法干其他的一些事情。怎么办?要重写 A1、A2、A3,并且加入在新模块中所需要的这些通用的方法吗?但是根据软件开发的接口依赖原则,新软件模块还能够依赖于抽象基类 A 吗?可是这些新增的通用方法并未在 A 中声明。或许应该考虑一下多重继承,将新增的通用方法抽象到一个新的接口 B 中,这样在使用 A1 的新增方法时,只需依赖于这个新的接口 B,而在使用 A1 以前的方法时,只需依赖于接口 A。

这是个好主意,于是新的派生类实现了。A11 继承于 A1 和 B, A22 继承于 A2 和 B, A33 继承于 A3 和 B。

13.6.1 多继承的实现

多继承可以看作是单继承的扩展。所谓多继承是指派生类具有多个基类，派生类与每个基类之间的关系仍可看作是一个单继承。多继承下派生类的定义格式如下：

```
class <派生类名>:<继承方式 1><基类名 1>,<继承方式 2><基类名 2>,...
{
    <派生类类体>
};
```

其中，<继承方式 1>、<继承方式 2>……是三种继承方式(public、private 和 protected)之一。例如：

```
class A
{
    ...
};
class B
{
    ...
};
class C : public A, public, B
{
    ...
};
```

13.6.2 多继承的二义性

多继承中的主要问题是标识不唯一。比如，在派生类继承的这多个基类中有同名成员时，派生类中就会出现来自不同基类的同名成员，就出现了标识不唯一或二义性的情况，这在程序中是不允许的。例如：

```
class base1
{
public:
    int x;
    int a();
    int b();
    int b(int);
    int c();
};
class base2
{
    int x;
```

```

    int a();
public:
    float b();
    int c();
};
class derived:base1,base2
{
};
void d(derived &e)
{
    e.x=10;           //错误, 不知道 x 是从哪个基类继承来的, 有二义性
    e.a();            //错误, 有二义性
    e.b();            //错误, 有二义性
    e.c();            //错误, 有二义性
}

```

解决这个问题的办法有三种, 一是使用作用域运算符“::”, 二是使用同名覆盖的原则, 三是使用虚函数。关于虚函数的有关问题将在后面讨论, 这里先介绍前两种方法。

1. 使用作用域运算符

如果派生类的基类之间没有继承关系, 同时又没有共同的基类, 则在引用同名成员时, 可在成员名前加上类名和作用域运算符“::”, 来区别来自不同基类的成员。例如, 将上例中的函数 `d(derived &e)` 改写如下, 就不会出现这个问题了:

```

void d(derived &e)
{
    e.base1::x=10;
    e.base2::a();
    e.base2::b();
    e.base1::c();
}

```

2. 使用同名覆盖的原则

在派生类中重新定义与基类中同名的成员(如果是成员函数, 则参数表也要相同, 参数不同的情况为重载)以隐蔽掉基类的同名成员, 在引用这些同名的成员时, 使用的就是派生类中的函数, 也就不会出现二义性的问题了。例如:

```

#include<iostream.h>
class base
{
public:
    int x;
    void show()
    {
        cout<<"This is base, x="<<x<<endl;
    }
}

```

```

    }
};
class derived: base
{
public:
    int x;                //同名数据成员
    void show()           //同名成员函数
    {
        cout<<"This is derived, x="<<x<<endl;
    }
};
void main()
{
    derived ob;
    ob.x=5;               //用同名覆盖原则引用派生类数据成员
    ob.show();            //用同名覆盖原则引用派生类成员函数
    ob.base::x=12;        //用作用域运算符访问基类成员
    ob.base::show();      //用作用域运算符访问基类成员
}

```

程序运行结果如下:

```

This is derived, x=12
This is base, x=5

```

13.6.3 多继承的构造顺序

在多继承的情况下, 派生类的构造函数格式如下:

```

<派生类名>(<总参数表>) : <基类名 1>(<参数表 1>), <基类名 2>(<参数表 2>), ...
<子对象名> (<参数表 n+1>), ...
{
    <派生类构造函数体>
}

```

其中, <总参数表>中各个参数包含了其后的各个分参数表。

多继承下派生类的构造函数与单继承下派生类构造函数相似, 它必须同时负责该派生类所有基类构造函数的调用。同时, 派生类的参数个数必须包含完成所有基类初始化所需的参数个数。

派生类构造函数执行顺序是先执行所有基类的构造函数, 再执行派生类本身的构造函数, 处于同一层次的各基类构造函数的执行顺序取决于定义派生类时所指定的各基类顺序, 与派生类构造函数中所定义的成员初始化列表的各项顺序无关。也就是说, 执行基类构造函数的顺序取决于定义派生类时基类在派生表中出现的顺序。可见, 派生类构造函数的成员初始化列表中各项顺序可以任意地排列。

13.7 虚 基 类

在介绍多继承的概念时已经知道，多继承中，要引用派生类的成员时，先是在派生类自身的作用域内寻找，若找不到，再到基类中寻找，这时，如果这些基类又有一个共同的基类，派生类访问这个公共的成员时，就有可能由于同名成员的问题而发生二义性。例如：

```
#include<iostream.h>
class base
{
protected:
    int x;
public:
    base()
    { x=1; }
};
class base1: public base
{
public:
    base1()
    { cout<<"constructing base1,x="<<x<<endl; }
};
class base2: public base
{
public:
    base2()
    { cout<<"constructing base2,x="<<x<<endl; }
};
class derived: public base1,public base2
{
public:
    derived()
    { cout<<"constructing derived x="<<x<<endl; }
};
void main()
{
    derived obj;
    return 0;
}
```

这是一个有问题的程序。表面看起来类 `base1` 和类 `base2` 是从同一个基类 `base` 派生来的，但它们对应的却是基类 `base` 的两个不同的拷贝。因此，当派生类 `derived` 要访问变量 `x`

时不知从哪条路径去寻找,从而引发了二义性问题。非虚基类的类层次图如图 13-4(a)所示。

虚基类就是为了解决这个问题而引入的。具体的做法是将公共基类声明为虚基类,这样这个公共基类就只有一个拷贝而不会出现二义性了。虚基类的类层次图如图 13-4(b)所示。



图 13-4 继承层次图

13.7.1 虚基类的定义

虚基类的声明是在派生类的声明过程中进行的,一般形式为:

```
class<派生类名>: virtual<派生方式><基类名>
```

虚基类关键字的作用范围和派生方式与一般派生类一样,只对紧跟其后的基类起作用。声明了虚基类以后,虚基类的成员在进一步派生过程中和派生类一起维护同一个内存拷贝。例如,使用虚基类改写上面的例子程序如下。

```
#include<iostream.h>
class base
{
protected:
    int x;
public:
    base()
    { x=1; }
};
class base1: virtual public base
{
public:
    base1()
    { cout<<"constructing base1,x="<<x<<endl; }
};
class base2: virtual public base
{
public:
    base2()
    { cout<<"constructing base2,x="<<x<<endl; }
```

```
};  
class derived: public base1,public base2  
{  
public:  
    derived()  
    { cout<<"constrcting derived x="<<x<<endl; }  
};  
void main()  
{  
    derived obj;  
    return 0;  
}
```

例子中，由于把公共基类 `base` 声明为类 `base1` 和类 `base2` 的虚基类，所以由类 `base1` 和类 `base2` 派生的类 `derived` 只有一基类 `base`，从而消除了二义性。

13.7.2 虚基类的构造函数和初始化

虚基类的初始化与一般多继承的初始化在语法上是一样的，但构造函数的执行顺序不同，其执行顺序如下。

- 虚基类构造函数的执行在非虚基类的构造函数之前。
- 若同一层次中包含多个虚基类，这些虚基类的构造函数按对它们说明的先后次序执行。
- 若虚基类由非虚基类派生而来，则仍然先执行基类的构造函数，再执行派生类的构造函数。

虚基类在使用中还应注意以下几个问题。

- 虚基类的关键字 `virtual` 与派生方式的关键字 `private`、`protected` 和 `public` 的书写位置无关紧要，可以先写虚基类的关键字，也可以先写派生方式的关键字。
- 一个基类在作为某些派生了的虚基类的同时也可作为另一些派生类的非虚基类。
- 虚基类构造函数的参数必须有最新派生出来的类负责初始化，即使不是直接继承也应如此。

本章小结

本章讲述面向对象的两个重要特性——多态与继承，着重阐述代码重用的方法、函数的覆盖方法、不同访问权限的数据在继承前后访问权限的变化，利用虚拟函数实现多态性。

习 题

在自己熟悉的操作系统上，利用熟悉的图像 API 实现一个模拟 Windows 的绘图程序，要求至少支持 5 种不同类型的图形绘制，利用到虚拟与多态技术。



第 14 章 运算符重载

本章内容：

- 运算符重载的概念。
- 运算符重载的实现。
- 常用运算符的重载。

重点：

单目运算符++的重载。

目的：

扩展自定义类型的应用灵活度，让自定义类型与标准类型支持相同的数据运算符。

作为优秀的面向对象语言，C++通过多种手段支持多态性的实现，运算符的重载也是其中之一。C++允许一个相同的运算符代表多个不同实现的功能，这就是运算符的重载。用户可以根据需要定义对自己的类进行运算符重载，下面开始学习运算符重载的知识。

14.1 为何要重载运算符

运算符重载增强了 C++语言的可扩充性。在 C++中，说明一个类就是说明了一个新类型。因此，类对象和变量一样，可以作为参数传递，也可以作为返回类型。

在基本数据类型上，系统提供了许多预定义的运算符，它们可以用一种简洁的方式工作，例如乘法运算符“*”。

例如：对于两个整型数，计算它们的乘积，其程序如下。

```
int x,y,z;  
z=x * y;
```

这就是表达两个整数相乘的方法，非常简单。

但是有些时候情况却不是这样，例如字符数组，由于数组的特性，除非初始化的时候直接给数组赋值，而在其他时候，却不能用“=”进行赋值，也不能用其他运算符进行运算。

例如：

```
char a[10] = "abcd";    //正确
char b[10] = "efgh";    //正确
char c[30];
c = "abcdefgh";         //错误
strcpy(c, "abcdefgh");   //正确
c = a+b;                //错误
strcpy(c,a);
strcat(c,b);            //正确
```

从上面代码可以看到，对数组的操作，很多时候代码很复杂，可读性不高。如果定义一个类，类中设置一个成员变量用来存储数组，并给该类增加一个对“+”或“=”的解释，则该类可支持运算符，就可以方便地使用“+”或其他的符号，达到简单地操作数组的目的。为了表达上的方便，希望已预定义的运算符，如“+”、“-”、“*”、“/”等，也可以在特定类的对象上以新的含义进行解释。如上面的字符串连接则希望可以用下面的表达式语句实现：

```
c=a+b;
```

这就需要重载运算符来解决。

14.2 运算符重载的实现

重载运算符可使类对象可以使用系统预定义的运算符，使得对于类对象的操作与内置数据类型一样直观方便。

运算符重载的实质就是函数重载：每个运算符对应各自的运算符函数，根据操作数的不同调用不同的同名函数。将指定的运算表达式转化为对运算符函数的调用，运算对象转化为运算符函数的实参。

运算符是在 C++ 内部定义的，具有特定语法规则，如参数说明、运算顺序和优先级等。运算符重载时，要注意该重载运算符的运算顺序和优先级不变。需要记住的是，运算符重载的目的是使得 C++ 代码更直观、更易读，因此在重载运算符时不要改变运算符的普遍语言意义。比如，++ 是基本类型的自增操作，如果重载该运算符，其最终目的也应是对用户自定义类型的对象进行自增操作，而不宜实现其他如写磁盘等功能。也不能为整数加法符号再定义一个新的意义，如果可以这样做，程序就会太混乱了。运算符的操作数数目也是不可以被改变的。

但是，并非所有的运算符都是可以重载的，要记住 ., *, :: 和 ?: 运算符不允许重载。

C++中允许重载的运算符如图 14-1 所示。

C++ 允许重载的运算符								
+	-	*	/	%	^	&		~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	>>=	<<=	==	!=
<=	>=	&&		++	--	[]	()	new
delete								

图 14-1 C++中允许重载的运算符

运算符重载一般采用如下两种形式：重载为成员函数形式和重载为友元函数形式。之所以选择这两种形式，是因为只有这两种函数才可以访问类中的私有成员。下面分别来看运算符成员形式和友元形式的实现。

14.2.1 成员函数运算符重载

成员函数形式重载的运算符在类体中被声明，声明方式同普通成员函数一样，只不过它的名字由关键字“operator”以及紧随其后的一个预定义运算符组成。说明为类的成员函数格式如下：

声明：<返回类型> operator <运算符>(<参数表>);
 定义：<返回类型> <类名>::operator <运算符>(<参数表>){}

例如：

```
myclass
{
public:
    myclass operator+(&myclass);           //声明
    ...
}
myclass myclass::operator+(&myclass)      //定义
{
    ...//重载操作
}
```

该运算符必须来自 C++预定义运算符的一个子集，也就是说已经存在的运算符。

以字符串类的“+”和“=”运算符重载为例，看一下运算符重载的方法。字符串类中有一个重要的成员，就是一个字符数组。在本例中，假设数组足够长而不会出现数组访问越界。

```
class String
```

```
{
protected:
    char s[200];
public:
    String();
    void Init(char t[]);
    String operator+(const String &t);
    String operator=(char t[ ]);
    void Display();
};

String::String()
{
}

void String::Init(char t[])
{
    strcpy(s,t);
}

String String::operator+(const String &t)
{
    String temp;
    strcpy(temp.s,s);
    strcat(temp.s,t.s);
    return temp;
}

String String::operator=( char t[ ])
{
    strcpy(s,t);
    return *this;
}

void String::Display()
{
    cout<<s<<endl;
}

void main()
{
    String a,b,c;
    a = "abcd";
    b = "efgh";
    c = a+b;
    c.Display();
}
```

程序运行结果如下:

abcdefgh

在该程序中，类 `String` 定义了两个成员函数(+, =)作为运算符重载函数。定义了重载运算符以后，程序中出现的表达式：

`a+b`

编译程序将会解释为：

`a.operator+(b)`

可以注意到该运算符重载函数仅有一个参数“b”。可见，当重载为成员函数时，双目运算符仅有一个参数；对于单目运算符重载为成员函数时，不能再显式说明参数。

14.2.2 友元函数运算符重载

上面介绍了如何重载运算符，使之成为成员函数。对于双目运算符，友元函数有两个参数；对于单目运算符，友元函数有一个参数。重载为友元函数的运算符重载函数的定义格式如下：

```
friend <返回类型> operator <运算符>(<参数表>)
```

下面用友元函数运算符重载重新实现 14.2.1 节中 `String` 类的例子，程序如下。

```
class String
{
    friend String operator+(const String &t1,const String &t2);
protected:
    char s[200];
public:
    String();
    void Init(char t[]);
    String operator=(char t[ ]);①
    void Display();
};
String::String()
{
}
void String::Init(char t[])
{
    strcpy(s,t);
}
String operator+(const String &t1,const String &t2)
{
```

① 需要注意的是“=”运算符，不能用静态函数的形式重载。

```

    String temp;
    strcpy(temp.s,t1.s);
    strcat(temp.s,t2.s);
    return temp;
}
String String::operator+(const String &t)
{
    String temp;
    strcpy(temp.s,s);
    strcat(temp.s,t.s);
    return temp;
}
String String::operator=( char t[ ])
{
    strcpy(s,t);
    return *this;
}
void String::Display()
{
    cout<<s<<endl;
}

void main()
{
    String a,b,c;
    a = "abcd";
    b = "efgh";
    c = a+b;
    c.Display();
}

```

该程序的运行结果与上例相同。程序中出现的：

a+b

编译程序解释为：

operator+(a, b)

14.3 单目运算符和双目运算符

尽管运算符重载可以有成员函数重载和友元函数重载两种形式，但并不是在任何地方这两种方式都可以被使用。对于不同的操作数的运算符，有不同的重载方式适合于它们。如果使用不当，结果会使程序变得毫无稳定性可言。

看下面的例子：

```
class String
{
public:
    // 构造函数的重载集合
    String( const char * = 0 );
    String( const String & );

    // += 运算符的重载集合
    String& operator+=( const String & );
    String& operator+=( const char * );

    // ==运算符的重载集合
    bool operator==( const char * ) const;
    bool operator==( const String & ) const;
    // ...
private:
    // ...
};
```

这里定义了两个重载版本的 String 类的等于比较运算符。第一个比较两个 String 类对象是否相等，第二个比较一个 String 类对象是否等于一个 C 风格的字符串。但是在使用它们的时候：

```
#include "String.h"
int main()
{
    String flower                                //设置 flower
    if ( flower == "lily" )                        //正常
        // ...
    else if ( "tulip" == flower )                 //编译错误
        // ...
}
```

在主函数中，等于比较运算符两次调用了重载运算符：

```
operator==(const char*)
```

但是第二次使用等于运算符却导致了一个编译错误，怎么会这样呢？

两次调用仅仅是比较的两个对象的前后顺序不同。问题就在于此，当运算符是双目运算符且被重载为成员函数时，只有在左操作数是该类类型的对象时，才能够使用作为类成员的重载运算符。这是因为对于成员运算符重载，如下的表达式：

```
flower == "lily"
```

实际上等于是被编译器重写为:

```
flower.operator==( "lily" )
```

在成员运算符重载的定义中通过指向对象本身的 `this` 指针可以引用左操作数 `flower`。

第二次调用的左操作数不是 `String` 类类型, 而是内置类型 `char*`。所以编译器试图找到对应 `char*` 的一个内置运算符, 它的两个操作数分别为一个 `char*` 字符串和一个 `String` 类对象。但是事实上并不存在这样的运算符, 所以编译器为 `main()` 中第二次使用等于运算符时就会产生一个错误信息。

但是可以用构造函数将一个 C 风格字符串隐式地转换为一个 `String` 类的对象。在上面 `String` 类的声明里存在这样的构造函数, 为什么编译器没有隐式地做如上的转换呢? 像这样的调用就是正确的:

```
if( String( "tulip" ) == flower )           //ok: 调用成员运算符
```

问题在于这样做会影响编译效率。如果编译器要做像上面那样的隐式转换, 为了给这个比较操作找到等于运算符, 编译器必须查找所有的类定义以找到所有能够把左操作数转换成某个类的构造函数, 然后再为每一个类类型找到相关的重载等于运算符, 看是否有一个能执行等于操作。接着, 编译器还需要判断哪一个构造函数和等于运算符的组合对于右操作数是最佳匹配。因此, 考虑到实际效率, 根本不可能要求编译器这样做。所以编译器只查找在左操作数的类中定义的成员重载运算符, 以及在其基类中定义的重载运算符。

如果将上例中的双目运算符重载为友元函数, 则程序改为:

```
class String
{
public:
    // 构造函数的重载集合
    String( const char * = 0 );
    String( const String & );

    // += 运算符的重载集合
    String& operator+=( const String & );
    String& operator+=( const char * );

    // == 运算符的友元重载声明
    friend bool operator==( const String &, const String & );
    // ...
private:
    // ...
};

bool operator==( const String &, const String & )
{
```

```

    //...
}

```

在上面的程序里，仅定义了一个友元运算符重载：

```
bool operator==( const String &,const String &)
```

就满足了 `char==String` 和 `String==char` 两种形式的调用，这里就是使用了构造函数的隐式转换。

在 `String` 类中，定义了参数为 `char` 类型的构造函数。根据以前的内容，这也可以作为由 `char` 到 `String` 类对象的转换。对于友元函数的调用，它的形式为：

```
friend bool operator==( const char &,const String &);
friend bool operator==( const String &,const char &);
```

编译器看到有 `String` 类的存在，便可以到 `String` 类的定义中找到合适的类型转换和与之匹配的运算符重载。

因此一般说来，单目运算符最好被重载为成员函数；对双目运算符而言，则最好被重载为友元函数，双目运算符重载为友元函数比重载为成员函数更方便适用。但是，有的双目运算符还是重载为成员函数为好，例如赋值运算符。因为，它如果被重载为友元函数，将会出现与赋值语义不一致的地方。C++语法规定不能或不应当重载为友元函数的运算符是：`=`，`()`，`[]`和`->`。

14.4 引用返回和值返回

以下示例用于实现复数类的运算符重载，包括四则运算符和复合赋值运算符。

```

class complex
{
public:
    //构造函数
    complex(){ real=imag=0; }
    complex(double r, double i)
    {
        real = r, imag = i;
    }
    //四则运算符重载
    friend complex operator +(const complex &c1, const complex &c2);
    friend complex operator -(const complex &c1, const complex &c2);
    friend complex operator *(const complex &c1, const complex &c2);
    friend complex operator /(const complex &c1, const complex &c2);

```



```

    friend void print(const complex &c);

    //混合赋值运算符重载
    complex& operator +=(complex &c);
    complex& operator +=(complex &c);
private:
    double real, imag;
};
complex& operator +=(complex &c)
{
    return complex(c.real+=c.real, c.imag+=c.imag);
}
complex& operator -=(complex &c)
{
    return complex(c.real-=c.real, c.imag-=c.imag);
}

void print(const complex &c)
{
    if(c.imag<0)
        cout<<c.real<<c.imag<<'i';
    else
        cout<<c.real<<'+'<<c.imag<<'i';
}

void main()
{
    complex c(5.0, -3.0);
    c+=c+=c;
    cout<<"c+=c+=c=";
    print(c);
    cout<<endl;
}

```

仔细观察在上个例子中+=运算符的重载:

```
String& operator+=( const String & );
```

是通过引用返回的,而-=运算符的重载:

```
bool operator==( const String & ) const;
```

是通过值返回的。前面复数类的例子里,“+,-,*,/”四则运算符的重载返回值方式是值返回,“+=”是通过引用返回。在运算符重载中返回值的选择是如何决定的呢?这是类的设计者一时的决定,还是设计上的必需要求?下面来分析运算符重载的返回值。

C++重载运算符的目的就是使用户自定义类型尽可能地 and 预定义类型的工作方式相

似。在不同运算符重载上的不同实现方法正是为了符合运算符在一般使用上的习惯和特性。

对于 `operator+=()` 这个运算符带有赋值的特性。`operator++()` 在进行加法操作后会修改它的一个参数。而大部分运算符是可以级联操作的，`+=` 也不例外，考虑一下 `+=` 操作符级联使用的情况：

```
int i = 5;
i += i += i
```

结果为 `i=20`。上次 `i+=` 的返回值作为下一次的参数，其返回的结果还会在下一级被修改，这就要求其返回值是左值，这个条件决定了它不能以值的方式返回。如果以值返回，则会有意想不到的错误产生。例如：在复数类的 `+=`，`-=` 运算符重载中：

```
class complex
{
public:
    //构造函数
    complex(){ real=imag=0; }
    complex(double r, double i)
    {
        real = r, imag = i;
    }
    //四则运算符重载，值返回
    friend complex operator +(const complex &c1, const complex &c2);
    friend complex operator -(const complex &c1, const complex &c2);
    friend complex operator *(const complex &c1, const complex &c2);
    friend complex operator /(const complex &c1, const complex &c2);
    friend void print(const complex &c);

    //混合赋值运算符重载，也以值返回
    complex operator +=(complex &c);
    complex operator -=(complex &c);
private:
    double real, imag;
};

complex operator +(const complex &c1, const complex &c2)
{
    return complex(c1.real + c2.real, c1.imag + c2.imag);
}
//...
complex complex::operator +=(complex &c)
{
    c.real+=c.real;
    c.imag+=c.imag;
    return complex(c);
}
```

```

}
//...
void print(const complex &c)
{
    if(c.imag<0)
        cout<<c.real<<c.imag<<'i';
    else
        cout<<c.real<<'+'<<c.imag<<'i';
}

void main()
{
    complex c(5.0, -3.0);
    complex temp;
    temp=c+c+c;                //级联加法
    cout<<"c+c+c=";
    print(temp);
    cout<<endl;
    cout<<"c=";
    print(c);
    cout<<endl;

    c+=c+=c;
    cout<<"c+=c+=c=";          //级联+=
    print(c);
    cout<<endl;
    cout<<"c=";
    print(c);
    cout<<endl;
}

```

这里，加法的实现仍然是使用值返回的形式，而加复合赋值运算符也使用值返回的形式，程序运行结果为：

```

c+c+c=15-9i
c=5-3i
c+=c+=c=10-6i
c=10-6i

```

显然，加法的结果完全正确，而复合加赋值运算符的结果不正确。因为+=重载函数中返回一个对象值，但这个对象值并非是 c 本身，而是一个在返回过程中由 c 复制出的临时对象的值，它是从形参 s 中拷贝而来，随后又进行了括号外的++操作，再次产生临时对象，将值赋给 c。所以 c 本身只进行了一次+=操作。

而对于一般意义上的加法操作，不会改变它的两个操作数中的任一个对象，而且它必

须生成一个临时对象来存放加法的结果，并将该结果对象以值的方式返回给调用者。如果+以引用返回，如下例：

```
complex& operator +(complex& s1, complex& s2)
{
    complex temp;
    temp.real=s1.real+s2.real;
    temp.imag=s1.imag+s2.imag;
    return temp;
}
```

尽管以上程序可以正确通过编译，且能够运行，但会产生奇怪的结果。例中的 temp 对象由+运算符函数的栈空间分配内存，受限于块作用域，引用返回导致了调用者使用这块会被随时释放的空间。

能否将结果对象从堆中分配，以此来避免上例的问题呢？例如：

```
RMB& operator +(RMB& s1, RMB& s2)
{
    unsigned int jf=s1.jf+s2.Jf;
    unsigned int yuan=s1.yuan+s2.yuan;
    return * new RMB(yuan, jf);
}
```

虽然它无编译问题，可以运行，但是该堆空间无法回收，因为没有指向该堆空间的指针，会导致内存泄露，程序不断做加法时，堆空间也在不断流失。

如果坚持结果对象从堆中分配，而返回一个指针，那样在应用程序中就要付出代价：

```
void fn(RMB& a, RMB& b)
{
    RMB* pc=a+b; //c=a+b;必须由此三条语句代替
    RMB c= *pc;
    delete pc;
}
```

通过值返回，将有一个临时对象在调用者的栈空间产生，它复制被调函数的 result 对象，以便参加调用者中的表达式运算，对于“c=a+b;”，则 a+b 的临时对象赋给 c，然后临时对象的作用域也结束了。

与 operator+=() 不一样，operator++() 在进行加法操作后会修改它的一个参数。考虑到操作符可以级联使用，这就要求其返回值是左值，这个条件决定了它不能以值返回。如果以值返回：

```
RMB operator ++(RMB& s)
{
```

```

    s.jf++;
    if(s.jf>=100)
    {
        s.jf-=100;
        s.yuan++;
    }
    return s;
}
//. . .
RMB a(2, 50);
c= a++; //ok
c= ++a; //ok, a 为 2.52
c=++ (++a); //error, a 为 2.53, 理应 2.54

```

因为++a 返回一个对象值，这个对象值并非 a 本身，而是临时对象的值，它从形参 s 中拷贝而来，随后又进行了括号外的++操作，再次产生临时对象，将值赋给 c。所以 a 本身只进行了一次++操作。

14.5 常见的运算符重载

14.5.1 赋值与比较运算符

=、!=和==这些运算符是用来在任何类中实现想拷贝值的赋值运算，如果在类中写了拷贝构造函数，那么必须得实现这些功能，再一次强调，这些功能是新增到类的公共部分，公共部分包含了这些运算符并显露它们。

所有的比较运算符都关系到赋值运算，通常会在用户的类中提供拷贝构造函数与赋值操作。一般来说，如果类支持赋值运算符，那么它也应当支持同等的比较对象的能力，实现!=与==操作。实际上，比较运算符的重载会带来很大方便。

例如：

```

//作为友元函数：
bool operator == (const MyClass& l, const MyClass& r)
{
    bool isEqual;
    //... test each member,
    // set isEqual.
    Return isEqual;
};

```

```

bool operator != (const MyClass& l, const MyClass& r)
{
    return ! (l == r);
}
//作为成员函数:
bool operator == (const MyClass& r)      const
{
    bool isEqual;
    //... test each member,
    // set isEqual.
    Return isEqual;
};
bool operator != (const MyClass& r) const
{
    return ! (*this == r);
}

```

14.5.2 递增与递减运算符

此前已经接触过递增运算符的重载，那时候并没有区分前递增与后递增的差别，在通常情况下是分别不出++a 与 a++的差别的，但它们之间的确存在明显差别。

递增、递减运算符(++，--)是单目运算符。它们又有前置与后置之别。为了区分前缀和后缀两种运算，将后缀运算视为双目运算符，表达式：obj++或 obj--被看作为：obj++0或 obj--0。在后递增、递减运算符重载函数的参数中多加如一个 int 标识，标记为后递增、递减运算符重载函数。

下面举一例子说明重载递增、递减运算符的实现。

```

#include <iostream>
using namespace std;
class counter
{
public:
    counter() { v=0; }
    counter operator ++();
    counter operator ++( int );
    void print() { cout<<v<<endl; }
private:
    unsigned v;
};

counter counter::operator ++()
{
    v++;
}

```



```

        return *this;
    }

    counter counter::operator ++(int)
    //后递增,int 在这里只起到区分作用,事实上并没有实际作用
    {
        counter t;
        t.v = v++;
        return t;
    }

    void main()
    {
        counter c;
        for(int i=0; i<8; i++)
            c++;
        c.print();
        for(i=0; i<8; i++)
            ++c;
        c.print();
    }

```

要注意前递增运算和后递增运算之间的差别主要为以下两点。

- 运算过程中,先将对象进行递增修改,而后返回该对象(其实就是对象的引用)的叫前递增(增量)运算。在运算符重载函数中采用返回对象引用的方式编写。
- 运算过程中,先返回原有对象的值,而后进行对象递增运算的叫后递增(增量)运算。在运算符重载函数中采用值返回的方式编写,重载函数的内部实现必须创建一个用于临时存储原有对象值的对象,函数返回的时候就是返回该临时对象。

14.5.3 数组存取标识符

由于 C 语言的数组中并没有保存其大小,因此,不能对数组元素进行存取范围的检查,这就导致无法保证给数组动态赋值不会越界。利用 C++ 的类可以定义一种更安全、功能更强的数组类型。为此,为该类型定义重载下标运算符“[]”。

使用[]来存取数组中的单个元素,需要重载它们的唯一情形:模拟数组。当重载这个运算符时需要明白两点:首先,[]是二元运算符,这意味着这个运算符只有一个参数,不能使用这个运算符来使用多维数组;其次,想在数组中使用任意类型的数据,这允许用户结合数组来创建任何类型。例如:

```
#include <iostream>
```

```
class CharArray
{
public:
    CharArray(int l)
    {
        Length = l;
        Buff = new char[Length];
    }
    ~CharArray() { delete Buff; }
    int GetLength() { return Length; }
    char & operator [](int i);
private:
    int Length;
    char * Buff;
};

char & CharArray::operator [](int i)
{
    static char ch = 0;
    if(i<Length&& i>=0)
        return Buff[i];
    else
    {
        cout<<"\nIndex out of range.";
        return ch;
    }
}

void main()
{
    int cnt;
    CharArray string1(6);
    char * string2 = "string";
    for(cnt=0; cnt<8; cnt++)
        string1[cnt] = string2[cnt];
    cout<<"\n";
    for(cnt=0; cnt<8; cnt++)
        cout<<string1[cnt];
    cout<<"\n";
    cout<<string1.GetLength()<<endl;
}
```

该数组类的优点如下。

- 其大小不必是一个常量。
- 运行时动态指定大小可以不用运算符 `new` 和 `delete`。
- 当使用该类数组作函数参数时，不必分别传递数组变量本身及其大小，因为该

对象中已经保存大小。

在重载下标运算符函数时应该注意以下几点。

- 该函数只能带一个参数，不可带多个参数。
- 不得重载为友元函数，必须是非 static 类的成员函数。

本章小结

本章讲述，面向对象编程的一个重要特点——运算符重载，利用成员与友元，分别实现不同目运算符的重载。通过重载运算符，让自定义的类支持需要的运算符，只有支持运算符重载，自定义的类型才能成为真正好用的类型，这正是“面向对象编程”与“基于对象编程”的最大区别。

习 题

1. 制作一个字符串处理类，使其能够完成字符串的存储与显示工作。
2. 重载构造函数，让字符串对象的初始化支持=。
3. 重载+号运算符，让字符串类对象支持连接。
4. 增加一个方法，让该字符串处理类能完成与 sprintf 类似的工作。



第 15 章 模 板

本章内容:

- 模板的概念。
- 函数模板。
- 类模板。

重点:

定义和使用类模板。

目的:

掌握模板的应用技巧, 简化程序设计的编码工作量。

模板是 C++ 中应用最为广泛的技术, 同时使用也很简单。它被大量地应用于 C++ 所提供的库功能实现上, 其中最著名的当属标准模板库(Standard Template Library, STL), 本书将在第 16 章专门讨论它。

15.1 为何需要模板

C++ 是一种强类型语言, 强类型语言所使用的数据都必须明确地声明为某种严格定义的类型, 并且在所有的数值传递中, 编译器都强制进行类型相容性检查。虽然强类型语言有力地保证了语言的安全性和健壮性, 但有时候, 强类型语言对于实现相对简单的函数似乎是个障碍。例如, 虽然下面的函数 `min()` 的算法很简单, 但是强类型语言迫使用户不得不为所有希望比较的类型都实现一个实例。

```
int min( int a, int b )           // 整型比较
{
    return a < b ? a : b;
}
double min( double a, double b ) // 双精度实型比较
{
    return a < b ? a : b;
}
```

在模板出现之前，有一种方法可以作为这个问题的解决方案，但是这种方法也很危险，那就是使用带参数宏(也叫作伪函数宏)。例如，把上面的 `min()` 函数用下面的宏来代替：

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

它的意义就是在以后每一个出现了 `min(a,b)` 的地方，都使用预先定义好的语句来替换它。这里就是用 “`(a) < (b) ? (a) : (b)`” 来替换。而宏是不会进行类型检查的。

虽然该定义对于简单的 `min()` 调用都能正常工作，但是在稍微复杂的调用下，它就有可能出现错误。这是因为，宏的工作只是简单地进行代码文本的替换。若定义了算平方的带参数宏如下：

```
#define square(A) A*A
```

则，如下的调用：

```
square(a+2)
```

会被替换成 `a+2*a+2`，实际计算顺序变成了 `a+(2*a)+2`。

正因为使用宏在功能上的不便和不进行类型检查的危险，C++提出了模板的概念。模板分为函数模板和类模板。

15.2 函数模板

15.2.1 函数模板的概念

函数模板提供了一种机制，通过它可以保留函数定义和函数调用的语义，在一个程序位置上封装一段代码，以确保在函数调用之前实参只被计算一次，而无须像带参数宏那样绕过 C++ 的强制类型检查。

函数模板提供一种用来自动生成面向各种类型参数的函数实例的算法，使用它的程序员需对函数接口参数和返回类型中的全部或者部分类型进行参数化(parameterize)，而函数体保持不变。如果一个函数的实现在一组实例上保持不变，并且每个实例都处理一种唯一的数据类型，如上面提到的大小比较函数 `min()`，那这个函数就是函数模板的最佳候选者。

模板和宏所做的工作在原理上都是文本替换。但是由于宏是由预处理器所作的简单文本替换，而模板是由编译器完成的替换工作，正是这样的差别决定了后者在功能与安全上远大于前者。

函数模板在安全上最大的改进就是模板函数和其他所有函数一样，参数只会被求值一

次，避免了 `min()`、`square()` 这样的函数在由宏实现时，由于递增、递减运算符的混合使用可能带来的错误。

另外，模板的类型是安全的，在模板被展开时已经明确地知道所有类型信息，而宏却做不到这一点。

15.2.2 函数模板的定义

有相同语义和功能实现，而仅仅是处理数据类型不同的函数可以被写为函数模板，这样就可以用不同的类型来实现对不同代码的重用，函数模板一般的定义如下：

```
template <模板类型参数表> FunctionName(形式参数表)
{
    ...//函数体定义
}
```

其中，模板类型参数表由 `class` 或 `typename` 关键字后加一个标识符构成，常使用 `class` 关键字。在函数类型参数表中这两个关键字的意义相同，它们表示后面的参数名代表一个潜在的内置类型或用户自定义类型，模板参数名由程序员选择，一般使用大写字母 `T`。例如，对于比较两个数大小的函数，写成函数模板为：

```
template <class T>
T min(T a, T b)
{
    return a < b ? a : b;
}
```

当 `T` 被声明为模板参数之后，就可以被当作一个内置类型指示符一样使用，模板类型参数可以出现在模板定义的余下部分，直到模板声明或定义结束。它的使用方式与内置的类型一样，可以用来声明变量、常量，作为函数参数和返回值。

这样的函数模板定义，编译系统不为其产生任何执行代码，只是对函数进行描述，表示它每次能单独处理在模板类型形式参数表中说明的数据类型。编译系统根据实际调用时提供的类型参数表中的类型，确认是否匹配函数模板中对应的形式参数表，然后生成一个实际的模板函数。

在函数模板定义中声明的对象或类型不能与模板类型参数同名，例如：

```
template <class T>
T min(T a, T b)
{
    int T;          //错误：重复声明模板类型参数
    return a < b ? a : b;
}
```

```
}
```

当然，模板类型参数表中的参数也是不可以重复的，例如：

```
template <class T, class T>      //错误
T min(T, T){...}
template <class T, class S>      //正确
T min(T, S){ ...}
```

15.2.3 函数模板的实例化

函数模板指定了函数除类型外的组成结构，用户可以根据一组实际类型或值构造出模板函数，这个构造过程被称为函数模板实例化(Template Instantiation)，它是隐式发生的，可以被看作是函数模板被调用或取函数模板地址的过程。

例如，在下面的程序中模板 min() 被实例化两次，一次是 int 类型的比较，另一次是 double 类型的比较。

```
template <class T>
T min(T a, T b)
{
    return a < b ? a : b;
}
int main()
{
    min( 10, 20 );           //生成的模板函数为 int min( int, int );
    min( 10.0, 20.0 );       //生成的模板函数为 double min( double, double );
    return 0;
}
```

15.2.4 重载模板函数

由函数模板实例化而来的模板函数可以像重载普通函数那样被重载。当同时定义有同名的函数模板和普通函数时，编译器会把这种情况当作重载来处理。

每当发现函数调用时，编译器首先寻找一个参数完全匹配的普通函数，如果找到了就调用它；如果失败，则寻找一个函数模板，使其实例化，产生一个完全匹配的模板函数，如果可以找到，就调用它；如果还是无法匹配，编译器会再尝试低一级的对函数重载的方法，例如通过类型转换可产生的参数匹配等，如果找到了匹配的函数，就调用它。例如：

```
T max(T a, T b)      //函数模板
{
    cout<< "this is template function! max is:"<endl;
    return (a>b)? a : b;
}
```

```
int max(int a, int b)          //重载的普通函数
{
    cout<< "this is overload function with int, int! max is:"<<endl;
    return (a>b)? a : b;
}
char max(int a, char b)
{
    cout<< "this is the overload function with int, char! max is: "<<endl;
    return (a>b)? a : b;
}
void main()
{
    int i=10, j=5
    char c1 = 'a', c2 = 'e';

    max(i, j);
    max(c1, c2);
    max(i, c1);
    max(c1, i);
}
```

程序运行结果如下:

```
this is the overload function with int, int! max is: 10
this is a template function! max is: e
this is the overload function with int, char! max is: a
this is the overload function with int, int! max is: 97
```

请注意在上面的函数中, 转化数据类型时要先转换第一个参数。

有时函数的名字与函数模板的名字相同, 但操作不同。利用编译程序在处理这种情况时, 首先匹配重载函数, 然后再寻求模板匹配的特性, 可以用重载的方法把它们区分开, 这种情况也是重载模板函数。例如:

```
template T max(T a,T b)
{
    return a>b?a:b;
}

char* max(char* a,char* b)
{
    return (strcmp(a,b)?a:b);
}

void main()
```

```
{
    cout <<"Max(\"Good\", \"Bad\") is " << max("Good", "Bad") << endl;
}
```

程序运行结果如下:

```
Max("Good", "Bad") is Good
```

编译器在看到 `max("Good", "Bad")` 调用时, 首先进行重载函数匹配, 结果匹配了非模板函数 `char max(char, char)`, 所以这里不会引发模板函数的产生。

15.3 类 模 板

15.3.1 类模板的定义

模板在 C++ 中更多的使用是在类的定义中, 最常见的就有 STL(Standard Template Library) 和 ATL(Active Template Library), 它们都是作为 ANSI C++ 标准集成在 VC 开发环境中的标准模板库。下面展示如何定义类模板, 类模板的一般定义格式如下:

```
template <class T> class 类名
{
    T member;          //T 类型的数据成员
    //...
}
```

例如:

```
template <class T>
class myclass
{
    T temp;
public:
    myclass(T data)
    {
        temp=data;
    }
    void display()
    {
        cout<<"T is"<<temp<<endl;
    }
};
```

通常在类的定义前面加上 `template<class T>`, 这样表示 T 就是这个类模板中的类型参

数，可以使用不同的类型赋予 T，从而得到不同的模板类。

注意成员函数 `display()`，在这里是用内联函数的形式实现的。那么，对于非内联的成员函数，它的定义在类模板的外面，于是就有了像下面这样子的，类模板的另一种声明方式：

```
template <class T>
class myclass
{
    T temp;
public:
    myclass(T name)
    {
        temp=name;
    }
    void display();
};
template <class T>
void myclass<T>::display()           //类声明以外的成员函数定义
{
    cout<<"T is"<<temp<<endl;
}
```

可以看出，这种定义方式与类的另一种定义方式很相似，所不同的也仅仅是几处而已。成员函数的实现上，要写出函数名的全部完整内容。首先，类体声明前面加上 `template <class T>` 的模板声明；其次，也是最容易忽略的，一定要注意类名，是 `myclass<T>` 而不是 `myclass`，然后才是成员函数名和函数体定义。

在类模板定义中，类模板的名字可以被用作一个类型指示符，凡是可以使用类名的地方都可以用它。例如，一个物品队列类，包含一个指向下一个物品的指针。

```
template <class T>
class ItemList
{
public:
    ItemList( const T & );
private:
    T item;
    ItemList *next;           //用类名声明指针
};
```

实际上，在类模板定义中 `ItemList` 类模板名的每次出现都是以下形式的缩写：

```
ItemList<Type>
```

这种简写形式只能被用在类模板自己的定义中，以及在类模板定义之外出现的成员定

义中。当类模板在其他位置中被用作一个类型指示符时，必须像上面成员函数定义时一样指定完整的模板参数表，就像 `myclass<T>`。

需要注意的是，类模板成员函数的定义必须与类模板定义在同一个文件中。因为，类模板定义不是类定义，编译器无法通过一般的手段找到类模板成员函数的代码，只有将它和类模板定义放在一起，才能保证类模板能正常使用。

15.3.2 使用类模板

类模板定义指定了怎样根据一个或多个实际的类型或值的集合来构造单独的类。类模板定义被用作类的特定类型实例的自动生成模板，例如当程序员这样写时：

```
myclass<int> a;
```

一个针对 `int` 型对象的 `myclass` 类就被从通用的类模板定义中创建出来的，从通用的类模板定义中生成类的过程被称为类模板实例化。例如，对于一个队列类模板：

```
template <class T>
class ItemList<T>
{
public:
    ItemList<T>();
    ~ItemList<T>();
    T& remove();           //从队列中取出元素
    void add( const T & ); //向队列中添加元素
    bool is_empty() const  //判断队列是否为空
    {
        return front == 0;
    }
private:
    T data;                //结点实际存储的内容
    ItemList *front;       //标记前后结点的指针
    ItemList *back;
}
```

当一个每个结点存储 `int` 型对象的 `ItemList` 类被实例化时，类模板定义中每次出现的模板参数 `T` 都被 `int` 取代，`ItemList` 类模板变成类：

```
class ItemList<int>
{
public:
    ItemList<int>();
    ~ItemList<int>();
    int& remove();
    void add( const int & );
```

```
    bool is_empty() const
    {
        return front == 0;
    }
private:
    T data;
    ItemList *front;
    ItemList *back;
};
```

同一个类模板针对不同类型参数生成的实例之间并没有什么联系，类模板的每个实例都构成一个独立的类类型。例如，int 型的 ItemList 实例没有权力访问 string 类型的 ItemList 实例的非公有成员。

类模板实例的名字是模板名<int>或模板名<string>，在类模板名后面的类型符号<int>或<string>被称为模板类型实参。多个模板类型实参必须由逗号分隔。

类模板的实例名必须总是显式地指定模板实参。因为，与函数模板使用相应的类型来被调用不一样，类模板的实参无法通过程序的上下文环境被判断出来。

类模板有了实例化的类型后，就要使用它来声明和定义对象使用。模板类对象的声明和使用方式与非模板类类型的对象相同，只不过它们的类型名像模板名<int>而已。例如：

```
extern Item<double> a;
Item<int> *p = new Item<int>;
Item<int> q[1024];
```

15.4 综合应用

回顾第 11 章中栈类的设计，虽然已经过合理的改造，但是并不代表当时的设计可以满足实际应用的需求。例如在实际工作中，栈中存放的数据随着应用的不同会不同。在第 11 章及其以前的章节中所设计的栈，工作重点在于实现栈的逻辑，但是对数据类型没有进行过多的考虑，当时假设存储的数据都是整型的数据，而在实际工作中，任何类型的数据都可能作为数据存入栈内，因此当时的设计不能满足更多的数据类型的要求。在没有学习模板之前，唯一的解决办法也只能是对可能用到的数据类型进行重新设计栈，但有了模板，情况就不一样了，可以设计一个模板类来解决问题。设计的思路如下。

- (1) 定义模板。
- (2) 将栈中的用于存储数据的指针指定为模板类型。
- (3) 修改压栈函数及弹栈函数，将操作数的类型也修改为模板类型。

经过这样的改造, 该栈可以存储任何数据类型(前提是, 数据类型支持栈操作中涉及的运算符, 例如 “=”)。

程序清单:

```
#include <iostream.h>
template <class T>
class Stack
{
public:
    Stack();
    void Push(T x);
    bool Pop(T &x);
protected:
    int Pos;
    T Data[200];
};
template <class T>
Stack<T>::Stack()
{
    Pos=0;
}
template <class T>
void Stack<T>::Push(T x)
{
    Data[Pos++]=x;
}

template <class T>
bool Stack<T>::Pop(T &x)
{
    if(Pos==0)
    {
        return false;
    }
    else
    {
        x=Data[--Pos];
        return true;
    }
}

void main()
{
    Stack<int> a;
    int m;
```



```
    if(a.Pop(m))  
    {  
        cout<<m<<endl;  
    }  
    else  
    {  
        cout<<"empty stack"<<endl;  
    }  
}
```

本章小结

本章讲述了模板的应用，分别讲述了模板函数与模板类的设计与应用。



第 16 章 标准模板库

本章内容:

- 标准模板库(STL)的基本组成。
- 基本的 STL 元素使用。

重点:

基本 STL 元素的应用。

目的:

掌握利用 STL 元素编写应用程序, 提高代码编写效率。

在学习标准模板库之前, 首先要了解什么是标准模板库。标准模板库对应的英文是 Standard Template Library 缩写为 STL, 它是一个 C++ 容器类库。从根本上说, STL 是一些“容器”类及其算法的模板的集合, 也就是可以包含其他对象的类的集合, 也包括操纵这些容器及容器中数据的方法的集合。它们中的每个类都是以模板的形式提供的, 能包含各种类型的对象。

STL 作为 ANSI C++ 标准的一部分, 被内建在 VC 编译器之内, 而不需要额外的安装。STL 的目的是提供标准化的组件, 这样用户就可以立即使用这些现成的组件, 而不用费时费力的重新为每一个程序写同样的代码了。

STL 中包括的容器类有 vector(向量类)、list(链表类)、deque(双向链表类)、set(集合类)和 map(映射类)等。同时, STL 也是算法和其他一些组件的集合。由于 STL 主要使用模板来实现, 模板也被叫做范型(generic), 类模板叫做范型类(Generic Class), 而函数模板也自然而然地被叫做范型函数(Generic Function), 因而使用 STL 的编程方法又被称作范型编程。

16.1 标准模板库的基本组成

容器、迭代器和算法是三类主要的 STL 组件。STL 的代码从广义上讲分为三类:

algorithm(算法)、container(容器)和 iterator(迭代器),几乎所有的代码都采用了模板类和模板函数的方式,这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。在 C++ 标准中,STL 被组织为下面的 13 个头文件: <algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack> 和 <utility>。下面就简单介绍一下 STL 各个部分的主要特点。

STL 容器是可以保存不同类型对象的模板类,也就是不同数据类型的“容器”,由它们来负责对象的存取、操作。不同容器定义了不同的操作容器对象的接口。

STL 迭代器也叫游标,就像是容器中用来指向对象的指针。STL 的算法使用 iterator 在容器上进行操作。iterator 也设置算法的边界、容器的长度和其他一些事情。举个例子,有些 iterator 仅允许算法读取元素,有一些允许算法写元素,有一些则两者都行。iterator 也决定在容器中进行处理的方向。用户可以通过调用容器的成员函数 begin()来得到一个指向容器起始位置的 iterator,也可以通过调用容器的 end()函数来得到上次操作的最后一个位置。

STL 算法就是标准算法,可以把它们应用在上面那些容器中的对象上。这些算法都有自己独特的功能,用户可以使用它们来完成对象排序、删除、记数、比较、特定对象的查找和合并容器等一些通用的操作。

算法以合适、标准的方法操作对象,并可通过 iterator 得到容器精确的长度,并且控制算法在容器中的边界。当然,其中还有一些其他的,对这些核心组件类型有功能性增强作用的组件,例如函数对象。接下来将逐步了解如何使用 STL。

16.2 标准命名空间

与 STL 相关的一个概念是命名空间(namespace)。标准 C++ 库及 STL 中的所有组件都是在 std 命名空间中声明和定义的,如在标准头文件 <iostream> 或 <vector> 中声明的函数对象和类模板都是被声明在名字空间 std 中的。

名字空间 std 的成员不能被不加限定修饰地访问,为了修正这个错误可以选择下列 3 种方法,用来声明程序所使用的命名空间。

(1) 用 using 关键字将 std 这个名字空间引入全局名字空间,在文件顶部的位置,头文件包含声明的语句(形如“#include...”)下面加入:

```
using namespace std;
```

这对单个工程来说是最简单的方法,但这样做就失去了 namespace 用于解决名字空间污染问题的意义。

(2) 用 `using` 声明只引入代码中用到的、属于名字空间 `std` 的某个成员。

```
using std::cout;           //使用 cout 对象
using std::endl;          //使用 endl 对象
using std::flush;         //使用 flush 对象
```

这样写的缺点是有些繁琐，如果有很多需要声明的对象，这可会是个麻烦的工作。但好处是对所有使用的对象有清晰地掌握，并且还可以容易地声明并使用其他命名空间中的成员。

(3) 在每一次使用 `std` 命名空间中的成员时，使用 `std` 域标识符 “`std::`”。比如：

```
typedef std::vector VEC_STR;
```

这种方法虽然写起来比较冗长，但是是在混合使用多个命名空间时的最好方法。

16.3 容 器

`Container` 顾名思义，就是一种容器，数据的容器，里面可以装各式各样的数据。一般形式如下：

```
template <class T>
class container
{
    T d;
};
```

透过 `template<class T>`，`T` 可表示任意数据类型，于是在容器类别内产生一个 `T` 类型 `d`，不管是 C++ 内建的基本数据类型或是自定义的类，都可以存入 STL 的容器类中。

STL 容器按存取顺序大致分为两种，顺序容器 (Sequence Container) 与关系容器 (Associate Container)。顺序容器内的每一个元素只能包含一种类型的数据成员，且各元素的排列顺序完全依照元素插入时的顺序。关系式容器内的每一个元素则包含两部分，一部分是键值，另一部分为数据本身。也就是说，关系容器内元素的排列及存取顺序方式是按键存取的。

16.3.1 顺序容器

顺序容器就是由同一类型元素依顺序组成的一个排列集合。顺序容器有以下几种基本的容器类别：向量 (vector)、双向队列 (deque) 和链表 (list)。

1. 向量

向量，或称为矢量，相当于基本数据类型中的数组。它就像是一个功能加强的队列，不但能以队列形式进行索引，还支持元素动态的添加和扩充，它是 C++ 标准容器模板类中的大哥大，使用十分广泛。

向量把被包含的对象以类似数组的形式连续存储，支持下标索引形式的访问，因此访问速度最快。但由此而产生的问题在于，由于数据存储形式的固定化，当用户希望在 `vector` 中间部位插入对象的时候，效率十分低下。因为它所分配的连续空间，使插入变成了多次的数据移动操作。

一个简单的 `vector` 容器使用如下。

```
#include <iostream>
#include <vector>
using namespace std;
void main()
{
    vector<char*> v;
    v.push_back("first");
    v.push_back("second");
    v.push_back("three");
    v.push_back("four");
    for(int i=0;i<4;i++)
        cout<<v[i]<<"\n";
}
```

程序的输出如下：

```
first
second
three
four
```

`vector<data_type>` 表示可以放入各种类型名称到 “<>” 之中，`vector` 容器便会正确地产生空间存放此类型的数据。`v.push_back(object)` 函数则把属于该类型的数据存入容器空间中。

一个 `vector` 容器好比一个传统 C 语言中的数组，只是传统 C 语言中的数组必须明确地指出大小，如果使用时下标值超过边界值，则系统将会产生未知的错误且没有警告。然而，STL 的 `vector` 容器却不然，`vector` 容器可以视需要自动增加大小，所以程序员就不需担心超过数组范围的非法存取的发生。

`start` 表示阵列起始处，`finish` 表示阵列结束处，`end_of_storage` 表示实际阵列结束的地方，如果阵列仍持续成长，超过 `end_of_storage`，则 `vector` 容器会自动再配置一块记忆体，维持下标值不超过边界值。

2. 双向队列

英文名为“double-ended-queue”，这是 C++ 顺序容器中闻名遐迩的双向队列。它在设计之初，就为从两端添加和删除元素做了特殊的优化。同样也支持随即访问，也有类似于 vector 的“[]”操作符，但大家不要因此就把它和 vector 混为一谈。

特点：从本质上讲，deque 在分配内存的时候使用了 map 的结构和方法，化整为零，分配了许多小的连续空间，因此，从 deque 两端添加、删除元素是十分方便的。最重要的一点是，如果在不知道内存具体需求的时候，使用 deque 绝对比 vector 好。大多数情况下，deque 和 vector 是可以互换使用的。

deque 容器如同数据结构中的双向队列，单向队列具有 FIFO(先进先出队列)的性质，双向队列则更进一步，可以由两边存入数据。

3. 链表

链表是指模板中的双向链表，设计它的目的可能就是为了在容器中间实现插入、删除，它的随机访问速度缓慢，而且没有“[]”操作。

特点：随机地插入、删除元素，在速度上占有明显的优势。并且，由于内存分配不连续，它对插入的要求也十分的低。所以在使用大对象的时候，这是一个不错的选择。list 容器为一个双向列表，可以用作树等结构。

16.3.2 关系式容器

关系式容器让 C++ 程序可以处理关系式数据。关系式数据是指一组数据由键值与数据成员所组成。如同数据库对数据表格的定义，STL 的关系式容器中的键值可以选择是否重复，不可重复的关系式容器好比数学中的集合概念，集合内的数据皆是唯一性的。由于这些容器在存入数据成员时都会把新的数据成员依键值做排序，所以可想象成好比一个有序的表或集合，甚至关系式容器也容许数据成员是空的，容器只储存键值，并做排序而已。

STL 共定义了 set<Key>、multiset<Key>、map<Key,T>和 multimap<Key,T>四种关系式容器，其主要特色为：物件带有关键值。各种容器的特色如表 16-1 所示。

表 16-1 各种容器特色

容 器 名	键值可重复吗	容器有数据成员吗
set<Key>	否	否
multiset<Key>	是	否
map<Key,T>	否	是
multiset<Key,T>	是	是

关联容器支持查询一个元素是否存在，并且可以有效地获取元素。两个基本的关联容器类型是映射(map)和集合(set)。映射是一个键/值对，键用于查询而值包含用户希望使用的数据。例如，可以很好地支持电话目录，键是人名，值是相关联的电话号码。

集合包含一个单一键值，可有效支持关于元素是否存在的查询。例如要创建一个单词数据库且它包含在某个文本中出现的单词时，文本查询系统对能生成一个单词集合以排除 the and 以及 but 等，程序将顺次读取文本中的每个单词，检查它是否属于被排除单词的集合并根据查询的结果将其丢弃或者放入数据库中。

映射和集合都只包含每个键的唯一出现，即每个键只允许出现一次。多映射和多集合支持同一个键的多次出现。例如，一个电话目录可能需要为单个用户支持多个列表，一种实现方法是使用多映射。

一个简单使用 map 容器的程序如下。

```
#include <iostream.h>
#include <iomanip.h>
#include <cstring.h>
#include <map.h>
void main()
{
    map<string,int,less<string>> m;
    m["me"]=1;
    m["you"]=2;
    m["he"]=3;
    m["she"]=4;

    cout<<setw(3)<<m["she"]<<setw(3)<<m["he"]<<setw(3)<<setw(3)<<m["you"]
        <<setw(3)<<m["me"];
}
```

程序输出如下：

```
4  3  2  1
```

less<string>为一个比较类别，目的在于如何在众多对象中(string 对象)，知道哪一个对象优先权高、哪一个低。而键对象是指“me”、“you”等字符串，数据成员对象是指 1、2、3、4 等整数对象。

由表 16-1 可以知道关系式容器的每个元素最多可含两种对象。只含有键对象的统称 set 类对象；含有键对象与值对象的统称 map 类对象。而每类对象又可依其是否可含重复键，分成 single 对象与 multi 对象。这些对象事实上都是数据结构中的树状结构。每个容器在存入对象时，便依键对象的大小决定它在树中的位置。而程序中 less<string>是后段提到的

function 对象，在此先略过不谈。

16.4 迭 代 器

迭代器提供对一个容器中对象的访问方法，并且定义了容器中对象的范围。迭代器就如同一个指针。事实上，C++的指针也是一种迭代器，但是，迭代器不仅仅是指针。

迭代器有各种不同的创建方法。程序可能把迭代器作为一个变量创建。一个 STL 容器类可能为了使用一个特定类型的数据而创建一个迭代器。作为指针，必须能够使用*操作符类获取数据，还可以使用其他数学操作符如++。典型的++操作符用来递增迭代器，以访问容器中的下一个对象。如果迭代器到达了容器中最后一个元素的后面，则迭代器变成 past-the-end 值。使用一个 past-the-end 值的指针来访问对象是非法的，就好像使用 NULL 为初始化的指针一样。

所有的 STL 迭代器共分为五类，各种迭代器组件简介如下。

1. 输入迭代器

输入迭代器(Input Iterators): 输入迭代器是最普通的类型，负责从指定串流读入数据，如同文件 I/O，提供对数据的只读访问。输入迭代器至少能够使用“==”和“!=”测试是否相等；使用“*”来访问数据；使用++操作来递推迭代器到下一个元素或到达 past-the-end 值。

为了理解迭代器和 STL 函数是如何使用它们的，现在来看一下 find()模板函数的定义：

```
template <class InputIterator, class T>
InputIterator find( InputIterator first, InputIterator last, const T& value)
{
    while (first != last && *first != value) ++first;
    return first;
}
```

2. 输出迭代器

输出迭代器(Output Iterators): 负责输出数据对象于指定的串流，如同文件 I/O，提供对数据的只写访问。

以上两个迭代器只是单纯地负责数据对象输入与输出的工作，在整个迭代器阶层中显得不太重要。

3. 前向迭代器

前向迭代器(Forward Iterators)如同简单的指针，支持 operator*()运算符和 operator++()

运算符，也就是说限制了前向迭代器进行的方向永远往前，并且不能跳格。它是最常用的迭代器，提供读写操作。

4. 反向迭代器

反向迭代器(Bidirectional Iterators)同前向迭代器前进方向相反，支持 `operator--()` 的功能。STL 顺序容器皆支持这个迭代器，因此反向迭代器可操作的容器物件最多。但是，就是因为支援的容器物件较多，所以执行速度比随机访问迭代器慢。反向迭代器提供读写操作，并能向前和向后操作。容器类 `list` 只支持反向迭代器。

5. 随机访问迭代器

随机访问迭代器(Random Access Iterators): 功能最强大的迭代器，几乎与普通指针一样，也支持 `operator[]()` 运算符的，但支持的容器只有 `vector` 与 `deque`。它提供对数据元素的读写操作，并能在数据序列中随机移动。例如：

```
// test program for iterator
#include <iostream>
#include <deque>

using namespace std;

void main()
{
    deque<int> x;
    x.empty()?cout<<"x 目前是空的"<<endl:cout<<"x 不是空的"<<endl;
    x.push_back(1);
    x.empty()?cout<<" x 目前是空的"<<endl:cout<<" x 不是空的"<<endl;
    x.push_back(2);
    x.push_back(3);
    x.push_back(4);
    for(deque<int>::iterator i=x.begin();i != x.end(); i++)
    {
        cout<<*i<<endl;
    }
}
```

`deque<int>` 容器定义为可储存 `int` 类型的数据，存入一些 `int` 值后，可以根据需要浏览之。声明 `deque<int>::iterator i`，表示 `i` 是为 `deque` 定义的一个迭代器，想象 `i` 为一个在 `deque` 容器上的指针，`q.begin()` 和 `q.end()` 方法分别返回了 `deque` 容器的开始与结束位置，通过 `++` 操作来移动 `i` 的位置；使用 `*i` 操作，取得 `i` 此时所指的容器内元素的值。



注意： 在 `find()` 算法中，注意如果 `first` 和 `last` 指向不同的容器，该算法可能会陷入死循环。

16.5 算 法

大家都能取得的一个共识是函数库对数据类型的选择对其可重用性起着至关重要的作用。举例来说，一个求方根的函数，在使用浮点数作为其参数类型的情况下，其可重用性肯定比使用整型作为它的参数类型要高。而 C++ 通过模板的机制允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化的时候。STL 就利用了这一点提供了相当多有用的算法。它是在一个有效的框架中完成这些算法的——可以将所有的类型划分为少数的几类，然后就可以在模板的参数中使用一种类型替换掉同一种类中的其他类型。

STL 提供了大约 100 个实现算法的模板函数，比如算法 `for_each` 将为指定序列中的每一个元素调用指定的函数，`stable_sort` 以用户所指定的规则对序列进行稳定性排序等等。这样一来，只要读者熟悉了 STL 之后，许多代码就可以被大大地化简，只需要通过调用一两个算法模板，就可以完成所需要的功能并大大地提升效率。

算法部分主要由头文件 `<algorithm>`、`<numeric>` 和 `<functional>` 组成。`<algorithm>` 是所有 STL 头文件中最大的一个(尽管它很好理解)，它是由一大堆模板函数组成的，可以认为每个函数在很大程度上都是独立的，其中常用到的功能范围涉及比较、交换、查找、遍历操作、复制、修改、移除、反转、排序和合并等。`<numeric>` 体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。`<functional>` 中则定义了一些模板类，用以声明函数对象。整个 STL 定义的演算法大致可分为七类，简介如下。

1. 不变序列算法

不变序列算法(Non-mutating sequence algorithms)只适用于循序式容器，并且执行完后不会改变容器内的值。如 `find()`、`count()` 函数等。

STL 的通用算法 `count()` 和 `count_if()` 用来给容器中的对象记数。就像 `for_each()` 算法一样，`count()` 和 `count_if()` 算法也是在 `iterator` 范围内来做的。例如，用 STL 的通用算法 `count()` 来统计 `list` 中的元素个数。在一个学生测验成绩的 `list` 中来数一数满分的个数。这是一个整型的 `list`。

```
#include <list>
#include <algorithm>
int main (void)
{
    list<int> Scores;
```

```
Scores.push_back(100); Scores.push_back(80);
Scores.push_back(45); Scores.push_back(75);
Scores.push_back(99); Scores.push_back(100);

int NumberOf100Scores(0);
count (Scores.begin(), Scores.end(), 100, NumberOf100Scores);

cout << "这里有" << NumberOf100Scores << " 个成绩是 100 的" << endl; }
```

程序的输出为:

这里有 2 个成绩是 100 的

`count()`算法统计等于某个值的对象的个数。上面的例子检查 `list` 中的每个整型对象是不是 100。每次容器中的对象等于 100, 它就给 `NumberOf100Scores` 加 1。

`count_if()`是 `count()`的一个更有趣的版本, 它采用了 STL 的一个新组件——函数对象。`count_if()`带一个函数对象的参数。函数对象是至少带有一个 `operator()`方法的类。有些 STL 算法作为参数接收函数对象并调用这个函数对象的 `operator()`方法。

函数对象被约定为 STL 算法调用 `operator` 时返回 `true` 或 `false`, 它们根据这个来判定函数。举个例子会说得更清楚些。`count_if()`通过传递一个函数对象来作出比 `count()`更加复杂的评估以确定一个对象是否应该被记数。

在这个例子里将数一数牙刷的销售数量。输入数据将提交包含四个字符的销售码和产品说明的销售记录。

```
#include <string>
#include <list>
#include <algorithm>

const string ToothbrushCode("0003");

class IsAToothbrush {
public:
    bool operator() ( string& SalesRecord ) {
        return SalesRecord.substr(0,4)==ToothbrushCode;
    }
};

int main (void) {
```



```

list<string> SalesRecords;

SalesRecords.push_back("0001 Soap");
SalesRecords.push_back("0002 Shampoo");
SalesRecords.push_back("0003 Toothbrush");
SalesRecords.push_back("0004 Toothpaste");
SalesRecords.push_back("0003 Toothbrush");

int NumberOfToothbrushes(0);
count_if (SalesRecords.begin(), SalesRecords.end(),
          IsAToothbrush(), NumberOfToothbrushes);

cout << "There were "
      << NumberOfToothbrushes
      << " toothbrushes sold" << endl;
}

```

程序运行结果为：

```
There were 2 toothbrushes sold
```

这个程序是这样工作的：定义一个函数对象类 `IsAToothbrush`，这个类的对象能判断出卖出的是否是牙刷。如果这个记录是卖出牙刷的记录的话，函数调用 `operator()` 方法返回一个 `true`，否则返回 `false`。

`count_if()` 算法由第一和第二两个 `iterator` 参数指出的范围来处理容器对象，它将对每个 `IsAToothbrush()` 返回 `true` 的容器中的对象增加 `NumberOfToothbrushes` 的值。

最后的结果是 `NumberOfToothbrushes` 这个变量保存了产品代码域为“0003”的记录的个数，也就是牙刷的个数。

注意：`count_if()` 的第三个参数 `IsAToothbrush()`，是由它的构造函数临时构造的一个对象。可以把 `IsAToothbrush` 类的一个临时对象传递给 `count_if()` 函数。`count_if()` 将对该容器的每个对象调用这个函数。

2. 变动序列算法

变动序列算法(Mutating sequence algorithms)只适用于循序式容器，并且执行完后会改变容器内的值。如 `copy()`、`swap()` 和 `fill()` 函数等。

3. 排序、搜索算法

排序、搜索算法(Sorting and searching algorithms)就如同其名字一样, 此类演算法执行排序与搜寻的工作, 只适用于循序式容器, 因为关系式容器必定是排序好的, 因此不需要此类函数。

4. 数值算法

数值算法(Numeric algorithms)执行一般简单的数值计算功能, 如 `accumulate()`、`partial_sum()`函数等。

5. 集合运算

集合运算(Set operations)算法适用于所有 STL 定义的容器物件, 执行集合的运算, 如 `set_union()`、`set_intersection()`和 `set_difference()`函数等。

6. 堆运算

堆运算(Heap operations)算法很像数据结构中堆的运算, 如 `push_heap()`、`pop_heap()`函数等, 与 Adaptor 容器物件很像, 只是前一个为资料结构定义其资料(Adaptor); 后一个为资料结构定义其操作函数(如 `xx_heap()`函数等)。

7. 其他算法

不属于上面分类的其他演算法, 如 `min()`、`max()`函数等。

本章小结

本章阐述了 STL 的定义、基本构成、容器与迭代器的使用和常用迭代器的基本使用技巧, 同时讲述了 STL 中常用的算法, 利用这些算法, 程序设计者可以用更少的代码完成更多的工作, 同时效率更高。为了更好地掌握相关内容, 读者应仔细阅读 STL 中基本的输入输出迭代器的代码。

第 17 章 I/O 流

本章内容：

- 流的概念及流类库。
- 输入输出的格式控制和 ios 成员函数。
- 输入输出运算符的重载。
- 文件的输入输出。

重点：

文件读写。

目的：

掌握输入输出操作的基本原理，理解标准 I/O 流的操作方法，通过 I/O 流操作文件。

C++完全支持 C 的输入输出系统，但由于 C 的输入输出系统不支持类和对象，所以 C++又提供了自己的输入输出系统，并通过重载运算符“<<”和“>>”来支持类和对象的输入输出。C++的输入输出系统是以字节流的形式实现的。

C++中的流是指数据从一个对象传递到另一个对象的操作。从流中读取数据称为提取操作，向流内添加数据称为插入操作。流在使用前要建立，使用后要删除。如果数据的传递是在设备之间进行，这种流就称为 I/O 流。C++专门内置了一些供用户使用的类，在这些类中封装了可以实现输入输出操作的函数，这些类统称为 I/O 流类。流具有方向性：与输入设备相联系的流称为输入流，与输出设备相联系的流称为输出流，与输入输出设备相联系的流称为输入输出流。

17.1 流 的 概 念

所谓流是指数据从一个位置流到另一个位置，如数据从键盘流入到程序中、数据从程序流向屏幕或文件。把数据的流动抽象为流。

将执行 I/O 操作的类体系称为 I/O 流类，实现流类的软件包称为流类库。C++的流类库

是用派生方法建立起来的输入输出类库，它有两个平行的基类 `streambuf` 和 `ios` 类，其他的流类都是从这两个基类直接或间接派生的。这些标准流类对象都包含在头文件 `iostream.h` 中，使用时应包含该头文件。

1. streambuf 类

`streambuf` 类是带有缓冲区的流类库，其作用如下。

- 提供物理设备的接口、缓冲或处理流的通用方式，几乎不需要任何格式。
- 提供对缓冲区的低级操作，如设置缓冲区，对缓冲区指针进行操作，从缓冲区取字符，向缓冲区存储字符等。
- 用作流类库中的基类，派生以下三个流类。
 - ◆ `filebuf` 类：使用文件来保存缓冲区中的字符序列。
 - ◆ `strstreambuf` 类：扩展类 `streambuf` 的功能，提供在内存进行提取和插入操作的缓冲区管理。
 - ◆ `conbuf` 类：扩展类 `streambuf` 的功能，用于处理输出、提供控制光标、设置颜色、定义活动窗口、清屏和清一行等功能，为输出操作提供缓冲区管理。

该类使用的缓冲区由一个字符序列和输入缓冲区指针与输出缓冲区指针组成，指针指向字符被取出或插入的位置。通常情况下，均使用这三个派生类，很少直接使用 `streambuf` 类。

2. ios 类

`ios` 类及其派生类为用户提供了使用流类库的接口，它们均由一个指针指向 `streambuf` 类。`ios` 类及其派生类使用 `streambuf` 及其派生类来完成对错误的格式化输入输出的检查，支持对 `streambuf` 类的缓冲区进行 I/O 时的格式化或非格式化转换。`ios` 作为流类库中的基类，可以派生出许多类，其类的层次关系如图 17-1 所示。

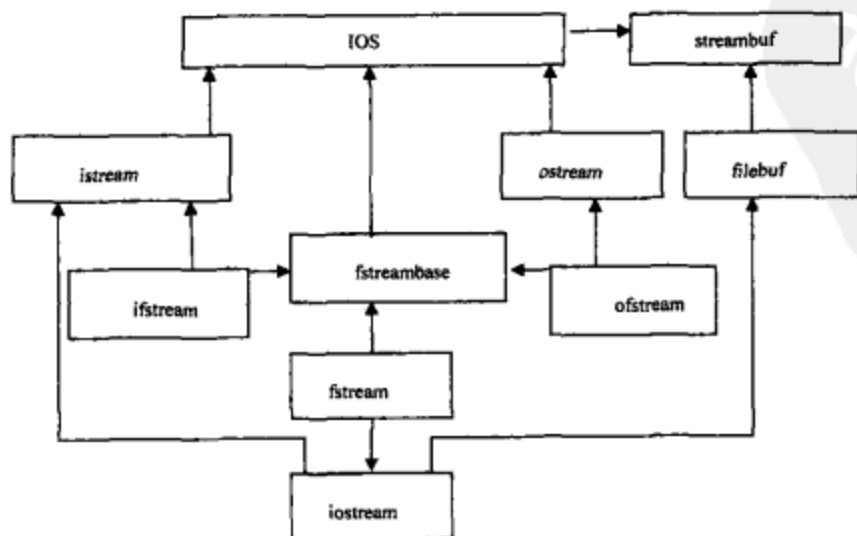


图 17-1 I/O 流类库的关系结构

17.2 I/O 标准流类

17.2.1 标准流的设备名

C++将一些常用的流类对象，如键盘输入、显示器输出、程序运行出错输出和打印机输出等，实现定义并内置在系统中，供用户直接使用。这些系统内置的用于设备间传递数据的对象称为标准流类对象，共有四个，如表 17-1 所示。

表 17-1 标准 I/O 流设备

C++中的名字	默认设备	C 中的名字	默认的含义
cin	键盘	stdin	标准输入
cout	屏幕	stdout	标准输出
cerr	屏幕	stderr	标准错误
clog	打印机	stdprn	打印机

在缺省方式下，标准输入设备是键盘，标准输出设备是显示器，而不论何种情况，标准输出设备总是显示器。cin 对象和 cout 对象前面已作过说明，cerr 对象和 clog 对象都是输出错误信息，它们的区别是：cerr 没有缓冲区，所有发送给它的出错信息都被立即输出；clog 对象带有缓冲区，所有发送给它的出错信息都先放入缓冲区，当缓冲区满时再进行输出，或通过刷新流的方式强迫刷新缓冲区。由于缓冲区会延迟错误信息的显示，所以建议使用 cout 对象。

应注意的是，cout 对象也能输出错误信息，但当用户把标准输出设备定向为其他设备时，cerr 对象仍然把信息发送到显示器。

17.2.2 原理

cout 是 ostream 流类的对象，它在 iostream 头文件中作为全局对象定义：

```
ostream cout(stdout); //标准设备名作为其构造时的参数
```

ostream 流类对应每个基本数据类型都有友元操作符“<<”，它们在 iostream 头文件中被声明，例如：

```
ostream& operator<<(ostream& dest, char* pSource);
ostream& operator<<(ostream& dest, int source);
ostream& operator<<(ostream& dest, char source);
```

分析语句:

```
cout<<"My name is Jone";
```

其中, `cout` 是 `ostream` 对象, `<<` 是操作符, 右面是 `char*` 类型, 故匹配上面的

```
ostream& operator<<(ostream& dest, char* pSource);
```

操作符。它会将整个字符串输出, 并返回 `ostream` 流对象的引用。

如果是:

```
cout <<"this is" <<7;
```

则根据 `<<` 的运算优先级, 此句可以看作:

```
(cout <<"this is")<<7;
```

由于 “`cout<<"this is"`” 返回 `ostream` 流对象的引用, 与后面的 `<<7` 匹配了另一个 “`ostream& operator<<(ostream& dest, int source)`” 操作符, 结果会构成连续的输出。

同理, `cin` 是 `istream` 的全局对象, `istream` 流类也有若干个友元:

```
istream& operator>>(istream& dest, char* pSource);
```

```
istream& operator>>(istream& dest, int source);
```

```
istream& operator>>(istream& dest, char source);
```

除了标准输入输出设备, 还有标准错误设备 `cerr`。当程序测试并处理关键错误时, 若不希望程序的错误信息从屏幕显示被重定向到其他地方, 这时可以使用 `cerr` 流显示信息。例如, 下面程序在除法操作不能进行时显示一条错误信息。

```
#include<iostream>
using namespace std;
```

```
void fn(int a, int b)
```

```
{
```

```
    if(b==0)
```

```
        cerr <<"zero encountered. " <<"The message cannot be redirected";
```

```
    else
```

```
        cout <<a/b <<endl;
```

```
}
```

```
void main()
```

```
{
```

```
    fn(20,2);
```

```
    fn(20,0);
```

```
}
```

程序运行结果如下：

```
c>ch19_1>abc.dat
zero encountered.The message cannot be redirected.
```

文件 abc.dat 的内容为：

```
10
```

主函数第一次调用 `fn()` 函数时，没有碰到除 0 运算，得到文件的写内容 10；第二次调用 `fn()` 函数时，碰到除 0 运算，于是在屏幕上输出错误信息。这就是 `cerr` 与 `cout` 的不同，写到 `cerr` 上的信息是不能被重定向的，只能在屏幕上显示。

17.3 输入输出的格式控制

C++ 仍可使用 C 中的 `printf()` 和 `scanf()` 语句进行格式化控制，同时又提供了两种格式化控制的方法，一是使用 `ios` 类中有关格式控制的成员函数，二是使用被称为格式控制符的特殊类型的函数。

17.3.1 用 `ios` 类的成员函数进行格式控制

此种成员函数进行格式控制主要是通过对格式状态字、域宽、填充字符和输出精度的操作来完成的。

状态字也称状态标志字，其类型为 `long int` 型，它是在 `ios` 类的 `public` 部分定义了一个枚举，此枚举类型的每个成员分别定义状态字的一个位，每个位称为状态标志位。这个枚举的定义如下：

```
enum
{
    skipws      =0x0001  //跳过输入中的空白，用于输入
    left        =0x0002  //左对齐输出，用于输出
    right       =0x0004  //右对齐输出，用于输出
    internal    =0x0008  //在符号位和基指示符后填入字符，用于输出
    dec         =0x0010  //转换基数为十进制，用于输入或输出
    oct         =0x0020  //转换基数为八进制，用于输入或输出
    hex         =0x0040  //转换基数为十六进制，用于输入或输出
    showbase    =0x0080  //输出时显示基指示符，用于输入或输出
    showpoint   =0x0100  //输出时显示小数点，用于输出
    uppercase   =0x0200  //十六进制输出时，表示制式的和表示数值的一律为大写，用于输出
    showpos     =0x0400  //正整数前显示"+"符号，用于输出
    scientific  =0x0800  //用科学表示法显示浮点数，用于输出
}
```

```

    fixed      =0x1000    //用定点形式显示浮点数，用于输出
    unitbuf    =0x2000    //在输出操作后立即刷新所有流，用于输出
    stdio      =0x4000    //在输出操作后刷新 stdout 和 stderr，用于输出
}

```

这些枚举元素值的共同特点是：使状态标志字二进制表示中的不同位为 1，它们共同组成状态标志字，存放在数据成员 `long x_flags` 中。这些状态之间是或的关系，可以几个并存。

`ios` 类提供了几个用于控制输入输出格式的成员函数，其中的参数 `flags` 是状态控制字，存放在 `ios` 类中的数据成员 `long x_flags` 中。主要成员函数的使用方法如下。

(1) 设置状态标志：是指用函数 `setf()` 将状态字标志位置“1”，函数原型为：

```
long ios::setf(long flags)
```

使用时的一般调用形式为：

```
流对象.setf(ios::状态标志);
```

例如：

```

istream isobj;
ostream oobj;
isobj.setf(ios::skipws);
obj.setf(ios::right|ios::showpos|ios::dec);

```

(2) 清除状态标志：是指用函数 `unsetf()` 将某一状态标志置“0”，函数原型为：

```
long ios::unsetf(long flags)
```

使用时的一般调用形式为：

```
流对象.unsetf(ios::状态标志);
```

(3) 取状态标志：是指用函数 `flags()` 返回状态标志字，有带参数和不带参数两种形式，它们的函数原型为：

```

long ios::flags()
long ios::flags(long flags)

```

使用时的一般调用形式为：

```

流对象.flags();
流对象.flags(ios::状态标志);

```

不带参数时返回当前的状态标志字；带参数时将状态字设置为 `flags`，并返回设置前的状态标志字。`flags()` 与 `setf()` 函数的区别是：`setf()` 是在原有的基础上追加设置，不改变原有设置；`flags()` 使用新的设置覆盖原有的设置，改变了原有设置。

(4) 设置域宽。域宽是指输出字符的长度。域宽的设置用函数 `width()` 完成，有带参数和不带参数两种形式，它们的函数原型为：

```
long ios::width()
long ios::width(int w)
```

使用时的一般调用形式为：

```
流对象.width();
流对象.width(int w);
```

不带参数时返回当前的域宽值；带参数时将域宽值设置为 `w`，并返回设置前的域宽值。域宽值存放在 `ios` 类中的数据成员 `int x_width` 中。

(5) 设置显示精度：是指用函数 `precision()` 设置浮点数输出时的显示精度，函数原型为：

```
long ios::precision(int p)
```

使用时的一般调用形式为：

```
流对象.precision(int p);
```

设置的显示精度值存放在 `ios` 类中的数据成员 `int x_precision` 中。

(6) 填充字符：是指当输出值少于域宽时将剩余部分用设定的填充字符填满，缺省时的填充字符为空格。设置填充字符的函数原型有带参数和不带参数两种形式，它们分别为：

```
long ios::fill()
long ios::fill(char ch)
```

使用时的一般调用形式为：

```
流对象.fill();
流对象.fill(char ch);
```

不带参数时返回当前的填充字符；带参数时将填充字符设置为 `ch`，并返回设置前的填充字符。设置的填充字符存放在 `ios` 类中的数据成员 `int x_fill` 中。

注意，使用填充字符函数时，必须与设置域宽函数配合使用，否则没有意义。

以下示例使用成员函数设置状态字。

```
#include<iostream.h>
void dispflags(long f)
{
    long I;
    for(i=0x8000;i;i=i>>1)
    {
        if (i&f)
            cout<<"1";
    }
}
```

```

        else
            cout<<"0";
    }
    cout<<endl;
}
main()
{
    long f;
    dispflags(f);
    out.setf(ios::showpos|ios::scientific);
    f=cout.flags();
    dispflags(f);
    out.unsetf(ios::scientific);
    f=cout.flags();
    dispflags(f);
    f=cout.flags(ios::oct);
    dispflags(f);
    f=cout.flags();
    dispflags(f);
    return 0;
}

```

程序运行结果如下:

```

00100000000000001    //显示缺省状态下的状态标志
00101100000000001    //显示执行 setf() 函数后的状态标志
00100100000000001    //显示执行 unsetf() 函数后的状态标志
00100100000000001    //显示执行 flags(ios::oct) 函数前的状态标志
00000000000100000    //显示执行 flags(ios::oct) 函数后的状态标志

```

注意, flags(long flags)函数返回的是执行该函数前的状态标志。

以下示例使用成员函数控制输入输出格式。

```

#include<iostream.h>
main()
{
    cout<<"x_宽度="<<cout.width()<<endl;
    cout<<"x_补位符="<<cout.fill()<<endl;
    cout<<"x_小数点位="<<cout.precision()<<endl;
    cout<<123<<"    "<<123.456789<<endl;
    cout<<"-----"<<endl;
    cout.width(10);
    cout.precision(3);
    cout<<123<<"    "<<123.456789<<endl;
    cout<<"x_宽度="<<cout.width()<<endl;
    cout<<"x_补位符="<<cout.fill()<<endl;
}

```



```

    cout<<"x_小数点位="<<cout.precision()<<endl;
    cout<<"-----"<<endl;
    cout.width(10);
    cout.fill('*');
    cout<<123<<"    "<<123.456789<<endl;
    cout.width(10);
    cout.setf(ios::left);
    cout<<123<<"    "<<123.456789<<endl;
    cout<<"x_宽度="<<cout.width()<<endl;
    cout<<"x_补位符="<<cout.fill()<<endl;
    cout<<"x_小数点位="<<cout.precision()<<endl;
    return 0;
}

```

程序运行结果如下:

```

x_宽度=0
x_补位符=
x_小数点位=0
123    123.456789
-----
123    123.457
x_宽度=10
x_补位符=
x_小数点位=3
-----
*****123    123.457
123*****    123.457
x_宽度=10
x_补位符=*
x_小数点位=3

```

17.3.2 使用格式控制符进行格式控制

显然,使用成员函数控制输入输出格式时,每个函数的调用都要写一条语句,它们还不能直接嵌入到输入输出语句中,使用很不方便。为此,C++又提供了另一种输入输出格式的控制方法,这就是使用称为格式控制符的特殊函数。

预定义的格式控制符可以直接嵌入到输入输出语句中,完成类似于 ios 类中控制输入输出格式的成员函数的功能。预定义的格式控制符如表 17-2 所示。

表 17-2 预定义的格式控制符

格式控制符	功 能
dec	以十进制形式输入输出整型数, 用于输入或输出
hex	以十六进制形式输入输出整型数, 用于输入或输出
oct	以八进制形式输入输出整型数, 用于输入或输出
ws	用于输入时跳过开头的空白符, 仅用于输入
endl	插入一个换行符并刷新输出流, 仅用于输出
ends	插入一个空字符, 用来结束一个字符串, 仅用于输出
flush	刷新一个输出流, 仅用于输出
setbase(int n)	把转换基数设置位 n(n=0,8,10,16), 缺省值为 0(十进制)
resetiosflags(long f)	关闭由参数 f 指定的格式标志, 用于输入或输出
setiosflags(long f)	设置由参数 f 指定的格式标志, 用于输入或输出
setfill(int c)	设置 c 为填充字符, 缺省为空格, 用于输入或输出
setprecision(int n)	设置小数位数, 缺省为 6 位, 用于输入或输出
setw(int n)	设置域宽, 用于输入或输出

格式控制符 `setiosflags(long f)` 和 `resetiosflags(long f)` 中的格式标志与 `ios` 类中控制输入输出格式的成员函数所用的标志基本相同, 这里不再说明。

预定义格式控制符分为带参数和不带参数的两种, 带参数的在头文件 `ios.h` 中定义, 不带参数的在头文件 `iostream.h` 中定义。使用它们时, 程序中应包含相应的头文件。

格式控制符被嵌入到输入输出语句中控制输入输出的格式, 而不是执行输入输出操作。

以下示例使用预定义的格式控制符控制输入输出格式。

```
#include<iostream.h>
#include<ios.h>
main()
{
    cout<<setw(10)<<987<<654<<<<endl;
    cout<<987<<setiosflags(ios::scientific)<<setw(15)<<987.654321<<endl;
    cout<<987<<setw(10)<<hex<<987<<endl;
    cout<<987<<setw(10)<<oct<<987<<endl;
    cout<<987<<setw(10)<<setbase(0)<<987<<endl;

    cout<<resetiosflags(ios::scientific)<<setprecision(3)<<987.654321<<endl;
    cout<<setiosflags(ios::left)<<setfill('&')<<setw(7)<<987<<endl;
    cout<<setiosflags(ios::right)<<setfill('#')<<setw(7)<<987<<endl;
    return 0;
}
```

```
}

```

程序运行结果如下：

```
987654
987  9.876543e+02
987      3db
3db      1733
1733      987
9.877e+02
987&&&&
####987

```

可见，格式控制符 `setw` 只对最靠近它的输出起作用；格式控制符 `dec`、`oct` 和 `hex` 的作用一直保持到重新设置为止；格式控制符 `setprecision` 在输出时作四舍五入处理。

17.4 输入输出运算符的重载

用户自定义数据类型的输入输出，是通过重载运算符 “<<” 和 “>>” 来实现的。

17.4.1 重载输出运算符

输出运算符 “<<” 也称为插入运算符，重载它用以用户自定义类型的输出。定义运算符 “<<” 重载函数的一般形式为：

```
ostream &operator<<(ostream &stream, 类名 对象名)
{
    //操作代码
    return stream;
}
```

其中，第一个参数 `stream` 是对 `ostream` 对象的引用，必须是输出流，它可以是其他合法的标识符，但必须与 `return` 后面的标识符相同；第二个参数接受将被输出的对象。

重载输出运算符 “<<” 时要注意以下两个问题。

- 重载输出运算符函数不能是类的成员函数，必须是类的非成员函数。
- 作为非成员函数，重载输出运算符函数不能访问类的私有成员，为解决这个问题，应把重载输出运算符函数定义为类的友元函数。

以下示例实现输出运算符 “<<” 重载。

```
#include<iostream.h>

```

```

class point
{
    int x,y;
public:
    point()
    { x=0; y=0; }
    point(int xx,int yy)
    { x=xx; y=yy; }
    friend ostream &operator<<(ostream &stream, point obj);
};
ostream &operator<<(ostream &stream, point obj)
{
    stream<<"x="<<obj.x<<"    "<<"y="<<obj.y<<endl;
    return stream;
}
main()
{
    point p1(1,2),p2(3,4);
    cout<<p1<<p2;
    return 0;
}

```

程序运行结果如下:

```

x=1    y=2
x=3    y=4

```

17.4.2 重载输入运算符

输入运算符“>>”也称为提取运算符,重载它用以用户自定义类型的输入。定义运算符“>>”重载函数的一般形式为:

```

istream &operator>>(istream &stream,类名 &对象名)
{
    //操作代码
    return stream;
}

```

其中,第一个参数 stream 是对 istream 对象的引用,必须是输出流,它可以是其他合法的标识符,但必须与 return 后面的标识符相同;第二个参数是一个引用,前面的“&”不能省掉。

重载输入运算符“>>”时要注意以下两个问题。

- 重载输入运算符函数不能是类的成员函数,必须是类的非成员函数。

- 作为非成员函数，重载输入运算符函数不能访问类的私有成员，为解决这个问题，应把重载输入运算符函数定义为类的友元函数。

以下示例实现输入运算符“>>”重载。

```
#include<iostream.h>
class point
{
    int x,y;
public:
    point()
    { x=0; y=0; }
    point(int xx,int yy)
    { x=xx; y=yy; }
    friend ostream &operator<<(ostream &output, point obj);
    friend istream &operator>>(istream &input, point &obj);
};
ostream &operator<<(ostream &output, point obj)
{
    output<<"x="<<obj.x<<"    "<<"y="<<obj.y<<endl;
    return output;
}
istream &operator>>(istream &input, point &obj)
{
    cout<<"请输入 x,y 的值: "<<endl;
    input>>obj.x;
    input>>obj.y;
    return input;
}
main()
{
    point p(10,20);
    cout<<p;
    cin>>p;
    cout<<p;
    return 0;
}
```

程序运行结果如下：

```
x=10    y=20
请输入 x,y 的值: 30  40
x=30    y=40
```

17.5 文 件 流

接下来从一个例子开始学习 C++文件流，第一个程序将创建一个文件，并向这个文件中写入一些字符，例子代码如下。

```
#include <fstream>
using namespace std;
void main()
{
    ofstream OutFile("myFile.txt");
    OutFile << "Hello,this is a ofstream.";
    OutFile.close();
}
```

运行这个程序，将在当前运行目录下创建一个名为 myFile.txt 的文件，内容为“Hello,this is a ofstream.”。下面逐步分析它是如何起作用的。

头文件包含语句#include <fstream>：如同使用标准输入输出流一样，在进行文件流操作时，需要包含此头文件。在这个头文件中声明了若干个文件输入输出流类，包括文件输入流 ifstream、文件输出流 ofstream 及文件输入输出流 fstream，它们都继承自 istream 和 ostream 类，继承关系如图 17-2 所示。

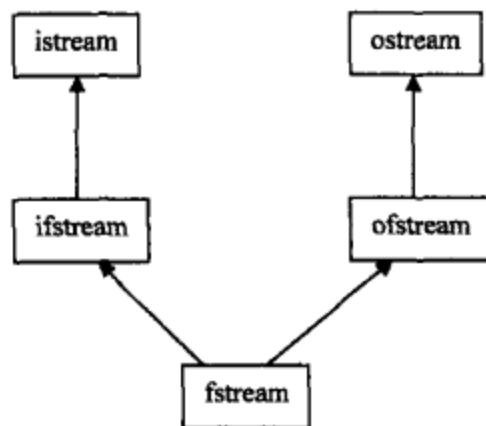


图 17-2 输入输出流的继承关系

ifstream 表示“input file stream(输入文件流)”。在前面的程序中出现的 ofstream，它的意义是“output file stream(输出文件流)”。进行文件的写操作，相对于文件来说就是“output(输出)”；读取一个文件，相对于文件来说就是“input(输入)”。

17.5.1 打开文件

不同于标准 I/O 对象，因为在 C++ 中文件类不是标准设备，所以没有预定义各自的全

局对象。因此文件 I/O 流对象要在使用时，由程序员来建立和初始化。初始化的内容包括规定文件名和文件的打开方式。创建文件流对象，还必须打开文件才可以访问文件、对文件流的对象进行操作。

打开文件有两种方法，在上面的例子中，通过创建 `ifstream`、`ofstream` 和 `fstream` 的对象，在创建时自动调用各自带参数的构造函数来打开文件，文件名和文件的打开方式就作为传递给构造函数的实参，例如：

```
fstream OpenFile(char *filename, int open_mode);
```

或者，对已有的流对象用 `open()` 成员函数来打开文件，于是上面的例子也可写为：

```
ofstream OutFile;           //创建对象
OutFile.open("myFile.txt"); //打开文件
...                          //写文件
```

上面的代码中，使用的是缺省参数的打开方式，创建 `ifstream` 对象缺省文件打开方式为输入方式，创建 `ofstream` 对象缺省文件打开方式为输出方式，而创建 `fstream` 的对象无缺省文件打开方式，必须指定打开方式。打开方式见表 17-3。

表 17-3 打开方式

名 称	描 述
<code>ios::in</code>	打开一个可读取文件
<code>ios::out</code>	打开一个可写入文件，如果文件存在，则清空文件内容；如果文件不存在则创建它
<code>ios::app</code>	你写入的所有数据将被追加到文件的末尾，此方式可使用 <code>ios::out</code>
<code>ios::ate</code>	你写入的所有数据将被追加到文件的末尾，此方式不可同时使用 <code>ios::out</code>
<code>ios::trunk</code>	删除文件原来已存在的内容(清空文件)
<code>ios:: Nocreate</code>	如果要打开的文件并不存在，那么以此参数调用 <code>open()</code> 函数将无法进行
<code>ios:: Noreplace</code>	如果要打开的文件已存在，试图用 <code>open()</code> 函数打开时将返回一个错误
<code>ios::binary</code>	以二进制的形式打开一个文件

实际上，以上的值都是一个枚举类型的 `int` 常量。下面是一个关于如何使用打开方式标志的例子。

```
#include <fstream>
using namespace std;
void main()
{
    ofstream SaveFile("file1.txt", ios::ate);
    SaveFile << "That's new added!\n";
    SaveFile.close();
}
```

正如表 17-3 中所显示的：使用 `ios::ate` 表示将会从文件的末尾开始执行写入。如果没有使用它，在默认的打开方式 `ios::out` 下原来的文件内容将会被重新写入的内容覆盖掉。

假如打算设置不止一个的打开模式标志，只需使用或操作符“|”，像这样：

```
ios::ate | ios::binary
```

下面就是怎样打开一个可以同时进行读取和写入操作的文件的例子。

```
fstream File("cpp-home.txt",ios::in | ios::out);
```

上面的代码创建了一个名为“File”的文件流对象，它是 `fstream` 类的一个对象。当使用 `fstream` 对象时，可以指定 `ios::in` 和 `ios::out` 作为文件的打开模式。这样，就可以同时对文件进行读、写，而无须创建新的文件句柄。当然，也可以只使用 `ios::in` 或者只使用 `ios::out`，那样的话，相应的只能进行读或者写的操作。

17.5.2 关闭文件

既然打开了一个文件流，那么当使用完它之后，就必须关闭它。对应于 `open` 函数，文件流类也有一个用于关闭文件的成员函数，即 `close()` 函数。只要调用这个函数，就可以关闭该流对象所打开的文件，调用格式如下：

```
OutFile.close();
```



注意：一旦关闭文件，在重新打开它以前，就再不能对它进行访问。这时，可以选择再次打开这个文件，或者打开另一个文件进行操作，或者释放流对象并结束程序。考虑以下的代码示例：

```
#include <fstream>
using namespace std;
void read(ifstream &T)           //将流对象作为参数传给函数
{
    char ch;
    while(!T.eof())
    {
        T.get(ch);
        cout << ch;
    }
    cout << endl << "-----" << endl;
}
void main()
{
    ifstream T("file1.txt");
    read(T);
    T.close();
}
```



```
T.open("file2.txt");
read(T);
T.close();
}
```

程序中的函数接受一个文件流对象，通过它来操作文件读出。在主函数中使用了同一个流对象，通过打开关闭不同的文件来完成对不同文件的读操作。

17.5.3 写入文件

有了可以在上面进行操作的对象，接下来就是如何读写文件了。“<<”看起来是不是很亲切？不错，在 `cout <<` 中见到过。这是一个预定义好的运算符。

```
OutFile << "Hello,this is a ofstream.";
```

这行语句所做的，是将 17.5 节开头那段文本写入文件。正如前面所提到的，`OutFile` 是一个文件流对象，它关联一个打开的文件。所以，只需输入“<<”，然后接着写下一串用引号括起来的文本，就可以实现对文件的写入。如果想写入的是某个变量的值而不是带引号的文本，也只需像通常使用 `cout` 一样将变量传递给流对象，像这样：

```
SaveFile << variablename;
```

17.5.4 读取文件

读文件和写文件是互逆的过程。现在，读取 `myFile.txt` 文件并将内容打印在屏幕上。下面是例子代码：

```
void main()
{
    ifstream inFile("myfile.txt");
    char ch;
    while(!inFile.eof())
    {
        inFile.get(ch);
        cout << ch;
    }
    inFile.close();
}
```

如果已经到达文件末尾，`eof()`成员函数将返回一个非零值。因此设计的这个循环将一直持续，直至该文件操作到达文件末尾。这样就可以遍历整个文件，以便对它进行读取。

`ifstream` 类声明了一个名为 `get()` 的成员函数。`get()` 函数从相应的文件流中依次读出一个

字符，并将其通过参数返回。在本例中，调用 `inFile.get(ch)` 后程序将会从 `inFile` 流中读取一个字符并存入变量字符 `ch` 中。当再次调用该函数时，它将接着读取下一个字符。这样，就不断反复循环直至读操作到达了文件结尾。每循环一次，读出的一个字符就被打印在屏幕上。

17.5.5 常用操作和状态检测

但是，并不是每次的打开文件操作都会成功，必须考虑如何检测文件打开操作是否成功。有几种方法可以实现，最简单而常用的方法是下面的例子：

```
fstream File("somedoc.txt");           //或 ofstream, ifstream
if (!File)
{
    cout << "打开文件失败\n";
    exit(1);
}
```

可以直接用逻辑符号检查文件流对象，来判断操作是否成功。或者，调用成员函数 `fail()`，代码如下。

```
ofstream File("file.txt", ios::nocreate);
if(File.fail())
{
    cout << "打开文件失败\n";
    exit(1);
}
```

`fail()` 函数用于测试是否有错误发生，如果有任何输入/输出错误(不包括到达文件末尾的情况)发生，它将返回非零值。

前面已经介绍过，可以创建一个流文件对象，但不立刻进行打开文件操作，像下面这样：

```
ifstream File;                          //也可以是一个 ofstream
```

如果在程序的某处，需要知道当前的流对象是否关联了一个已经打开的文件，那么可以用函数 `is_open()` 来进行检测。如果文件没有打开，它将返回 0；如果文件已经打开，它将返回 1。例如：

```
ofstream File1;
File1.open("file1.txt");
cout << File1.is_open() << endl;
```

上面的代码将会返回 1，因为在第二行，已经打开了一个文件 `file1.txt`。而下面的代码

则会返回 0，这是由于没有打开文件，而只是创建了一个流文件对象。

```
ofstream File1;
cout << File1.is_open() << endl;
```

C++中负责输入/输出的系统也包括了关于每一个输入/输出操作结果的记录信息。这些当前的状态信息被包含在 `io_state` 类型的对象中。像 `open_mode` 一样，`io_state` 也是一个枚举类型，表 17-4 是它包含的枚举值的名称及值相应含义的描述。

表 17-4 `io_state` 中的枚举值及其含义

枚举值名称	含 义
<code>goodbit</code>	无错误
<code>eofbit</code>	已到达文件尾
<code>failbit</code>	非致命的输入/输出错误
<code>badbit</code>	致命的输入/输出错误

有两种方法可以获得输入/输出对象的状态信息。一种方法是通过调用函数 `rdstate()`，它将返回当前状态的错误标记，即表 17-4 中的值。例如，假如没有任何错误，则 `rdstate()` 函数会返回标记 `goodbit`。例如，使用 `rdstate()` 的状态检测，代码如下。

```
// 实际应用中可将 FileStream 替换成相应在使用的文件流对象
if(FileStream.rdstate() == ios::eofbit)
    cout << "到文件结尾了!\n";
if(FileStream.rdstate() == ios::badbit)
    cout << "I/O 错误!\n";
if(FileStream.rdstate() == ios::failbit)
    cout << "I/O 错误!\n";
if(FileStream.rdstate() == ios::goodbit)
    cout << "没有错误!\n";
```

另一种方法则是使用下面任何一个函数来检测相应的输入/输出状态。

- `bool good();` 假如没有错误发生，`goodbit` 标志被设置，则 `good()` 函数返回 `true`。
- `bool eof();` 假如操作已经到达了文件末尾，`eofbit` 被设置，则 `eof()` 函数返回 `true`。
- `bool fail();` 假如 `failbit` 标志被设置，则 `fail()` 函数返回 `true`。
- `bool bad();` 假如 `badbit` 标志被设置，亦即出现了 `badbit` 对应的错误，`badbit` 状态被置为当前的错误状态，则 `bad()` 函数返回 `true`。

现在，检测到可能有错误发生，但是如果打算使程序能够继续正确地运行下去，那么就必须及时清除这些错误状态。要清除错误状态，需使用 `clear()` 函数。此函数带一个参数，就是将要为当前对象设置的标志值。假如想让程序继续运行下去，只要将 `ios::goodbit` 作为实参，`clear()` 函数示例代码如下。

```
void main()
{
    ofstream File1("file2.txt"); //建立 file2.txt
    File1.close();

    //因为使用了 ios::noreplace 打开模式，在试图打开一个已存在的文件时会返回错误
    ofstream Test("file2.txt",ios::noreplace);

    // 上一行将导致 ios::failbit 错误，下面就将其演示出来
    if(Test.rdstate() == ios::failbit)
        cout << "Error...\n";

    Test.clear(ios::goodbit); // 将当前状态重置为 ios::goodbit

    if(Test.rdstate() == ios::goodbit) // 检测程序是否已经正确地进行了设置
        cout << "Fine!\n";
    Test.clear(ios::eofbit); // 将状态标志设为 ios::eofbit
    if(Test.rdstate() == ios::eofbit) // 检测是否已经正确地进行了设置
        cout << "EOF!\n";
    Test.close();
}
```

除了使用标记值判断，也可以使用函数 `bad()`、`eof()`、`fail()`和 `good()`的形式进行判断，两者实际上是一样的——都是检测某个标记是否被设置。这些函数前面已经介绍过，这里就不再重复了。

17.5.6 读写随机文件

为了对文件进行随机操作，首先要知道如何才能在文件流中确定随机位置。在上节的例子中有过这样的代码：

```
while(!inFile.eof())
{
    inFile.get(ch);
    cout << ch;
}
```

这个 `while` 循环会一直反复，直至程序的操作到达文件的尾端。但这个循环如何知道是否已经到了文件末尾？

实际上，当 `get` 函数在读文件的时候，会有一个类似于“内置指针”的东西，它表明读取或写入已经到了文件的哪个位置，就像记事本中的光标。而每当调用 `inFile.get(ch)` 函数的时候，它会返回当前位置的字符，存储在 `ch` 变量中，并将这一内置指针向前移动一个字符。因此下次该函数再被调用时，它将会返回下一个字符。这一过程将不断反复，直到

读取到达文件尾。

所以，C++文件流对象也提供了成员函数，允许设定当前的读写位置，成员函数 `seekg()` 就是完成这个任务的，它将内置指针定位到指定的位置。可以使用以下标志作为参数：

```
inFile.seekg(ios::beg);    //ios::beg 标志将内置指针移动到文件首端
inFile.seekg(ios::end);    //ios::end 标志将内置指针移动到文件末端
```

或者，可以设定向前或向后跳转的字符数，将其跳转到相对于当前位置前或后的字符位置上去。例如，如果要想定位到当前位置的 5 个字符以前，应当写：

```
inFile.seekg(-5);
```

如果想向后跳过 40 个字符，则应当写：

```
inFile.seekg(40);
```

函数 `seekg()` 也有重载的版本，它可以带两个参数。另一个版本是这样子的：

```
File.seekg(-5,ios::end);
```

这个版本表示以第二个参数所标示的位置开始，由第一个参数所提供的偏移量确定将要移至的位置。下面是使用 `seekg()` 函数的示例。

```
void main()
{
    fstream File("test.txt",ios::in | ios::out);
    File << "Hi!"; //将"Hi!"写入文件
    static char str[10];
    File.seekg(ios::beg); // 回到文件首部
    File >> str;
    cout << str << endl;
    File.close();
}
```

`fstream File("test.txt", ios::in | ios::out);` 此行创建一个 `fstream` 对象，执行时将会以读/写方式打开 `test.txt` 文件。这意味着可以同时读取文件并写入数据。

`static char str[10];` 这行表示将创建一个容量为 10 的静态字符数组。声明为静态只不过是为了在创建数组的同时对其进行初始化。

在这个例子中，能够读到文件文本的最后 4 个字符，因为 `File.seekg(ios::beg);` 这行表示：

- 到达了末尾(`ios::end`)。
- 接着到达了末尾的前五个字符的位置(-5)。

为什么会读到 4 个字符而不是 5 个？只需把最后一个看成是“丢掉了”，因为文件最末

端的“东西”既不是字符也不是空白符，那只是一个位置。

当把“Hi”写进文件之后，内置指针将被设为指向其后面(也就是文件的末尾)。因此必须将内置指针设回文件起始处。这就是 `seekg()` 函数在此处的确切用途。

`File >> str;` 此行会从文件中读取一个单词，然后将它存入指定的数组变量中。

例如，如果文件中有这样的文本片段：

Hi! Do you know me?

使用 `File >> str`，则只会将“Hi!”输出到 `str` 数组中，因为它实际上是将空格作为单词的分隔符进行读取的。

由于上例存入文件中的只是单独一个“Hi!”，不需要写一个 `while` 循环，那样做会花更多的时间来写代码。这就是使用此方法的原因。顺便说一下，到目前为止，在读取文件的 `while` 循环中，程序读文件的方式是逐字符进行读取的。然而也可以逐单词进行读取，例如：

```
char str[30]; // 每个单词的长度不能超过 30 个字符
while(!OpenFile.eof())
{
    OpenFile >> str;
    cout << str;
}
```

当然也可以一行一行地进行读取，例如：

```
char line[100]; // 每个整行将会陆续被存储在这里
while(!OpenFile.eof())
{
    OpenFile.getline(line, 100); // 100 是数组的大小
    cout << line << endl;
}
```

在以上的读取方式中，建议使用逐行读取的方式，或者是最初的逐字符读取的方式。而逐词读取的方式并非一个好的方案，因为它不会读出换行信息，所以如果文件新起一行时，它不会将那些内容新起一行进行显示，而是加在已经打印的文本后面。而使用 `getline()` 或者 `get()` 函数才能完整保留格式的信息。

17.5.7 二进制文件的处理

虽然有规则格式的文本(到目前为止，所讨论的仅有文件形式.txt)非常有用，但有时候需要用到无格式的文件——二进制文件。这里所说的所谓“规则格式文本”即前面一直所使用的文本格式，而与之相对的“无格式文件”即是指存储各类数据或可执行代码的非文

本格式文件。对于文本文件，可以直接由预定好的“<<”及“>>”操作符进行读入/写出；对于二进制文件，通常需要把它逐字节读入内存，在二进制层次进行解析，与使用“<<”及“>>”操作符创建的文件大不相同。现在来讨论如何操作二进制文件。

首先是成员函数 `get()` 和 `put()`。成员函数 `get()` 与 `put()` 赋予了用户读/写无规则格式文件的能力：要读取一个字节，可以使用 `get()` 函数；要写入一个字节，则使用 `put()` 函数。前文我们曾经使用过 `get()` 函数。这里可能会疑问：为什么当时用户使用它时，输出到屏幕的文件内容看起来是文本格式的？这是因为在输出时仍然使用的是“<<”操作符，而不是 `put()` 函数。

成员函数 `get()` 与 `put()` 都各带一个参数：`get()` 参数为一个 `char` 型变量用于保存获取的字符；`put()` 的参数为一个字符常量或字符变量，函数将它输出在当前位置。

以上是对读写单个字节而言，假如要读/写一整块的数据，那么你可以使用 `read()` 和 `write()` 函数。它们的原型如下：

```
istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);
```

对于 `read()` 函数，参数 `buf` 是一个字符数组，由文件读出的数据将被保存在这儿；对于 `write()` 函数，`buf` 是一个字符数组，它用以存放要写入文件的数据。这两个函数中的 `num` 参数是指定要从文件中读取/写入的字节数的数字。

假如在读取数据时，在读取“`num`”个字节之前就已经到达了文件的末尾，那么可以通过调用 `gcount()` 函数来了解实际所读出的字节数。此函数会返回最后一次进行的对无格式文件的读入操作所实际读取的字节数。

在给出示例代码之前，需要补充的是，如果要以二进制方式对文件进行读/写操作，那么应当将 `ios::binary` 作为打开模式加入到文件打开的参数表中。

请看如下示例代码是如何运作的。

示例 1：使用 `get()` 和 `put()` 函数读写文件

```
#include <fstream.h>
void main()
{
    fstream File("test_file.txt", ios::out | ios::in | ios::binary);
    char ch;
    ch='o';
    File.put(ch);           // 将 ch 的内容写入文件
    File.seekg(ios::beg);   // 定位至文件首部
    File.get(ch);           // 读出一个字符
    cout << ch << endl;    // 将其显示在屏幕上
}
```

```
File.close();  
}
```

示例 2: 使用 read() 和 write() 函数读写文件

```
#include <fstream.h>  
#include <string.h>  
void main()  
{  
    fstream File("test_file.txt",ios::out | ios::in | ios::binary);  
    char arr[13];  
    strcpy(arr,"Hello World!"); //将 Hello World!存入数组  
    File.write(arr,5); // 将前 5 个字符——"Hello"写入文件  
    File.seekg(ios::beg); // 定位至文件首部  
    static char read_array[10]; // 在此打算读出些数据  
    File.read(read_array,3); // 读出前三个字符——"Hel"  
    cout << read_array << endl; // 将它们输出  
    File.close();  
}
```

本章小结

本章详细描述了 I/O 操作的基本流程,讲述了自定义类中重载输入输出运算符的方法,揭示了利用 I/O 流文件操作的基本方法。

习 题

1. 编写一个简单的文本文件查看器。
2. 编写一个图形文件浏览器,图形 API 可根据自己的情况自选。

第 18 章 异常处理

本章内容:

- 异常处理的概念和基本思想。
- C++异常处理的实现方法。
- C++异常处理中的构造与析构。

重点:

C++异常处理的实现方法。

目的:

掌握异常处理方法，提高程序的可读性。

从本质上看，程序异常是指出现了一些很少发生的或出乎意料的状态，通常显示了一个程序错误或要求一个必须提供的回应。不能满足这个回应经常造成程序功能削弱或死亡。有时候，异常导致整个系统崩溃。遗憾的是，试图使用传统的防护方法来编制健壮的代码经常只是将一个问题——程序的意外崩溃，换成了另外一个问题——更为混乱的设计和代码。

在 C++ 之前，太多的程序员认为这样做相比程序意外崩溃时造成的烦恼是得不偿失的，于是选择了生活在危险之中。认识到这一点后，C++ 标准增加了一个优雅并且基本上不可见的“异常体系”到语言中，来解决这个问题。就这样，异常处理方法产生了。

18.1 传统的错误处理方式

在早些时期，C++ 本身并没有处理运行期错误的能力。仅有的是那些继承自 C 的传统方法。这些方法可以被归纳为以下三类思想。

- 如果是函数，就让它自己返回一个状态码来表明成功或失败。
- 用一个全局标记作为错误码，将错误状态赋值给它并且让其他的函数来检测。
- 什么也不做，仅仅是终止整个程序。

上述的任何一个方法在面向对象环境下都有明显的缺点和限制。其中的一些根本就不可接受，尤其是在大型应用程序中。接下来的部分将会仔细探讨一下这些方法，以此来发现它们长处与限制和不足。

第一种返回错误码的方法在某种程度上是可以起作用的，比如一个小型程序，有着一致而且有限的错误码存在，并且严格的报告错误和检查一个函数的返回值策略被应用。然而，这种方法也有着明显的局限性。

首先，错误类型和它们的列举值必须标准化。因为一个库的实现者可能选择返回值 0 来代表一个错误，然而另一个实现者却可能选择 0 来代表成功并且用那些非 0 值代表出现错误。通常将返回码全部写在一个公共头文件中，以符号常量的形式存在，从而在整个软件的开发过程中或者在一个开发团队里达成一致。但是，C++标准中并没有这些码的定义。这样一来，在结合那些不兼容的软件库时，如何处理非标准的错误码将会是一件极其头疼的事。

另外，对于每一个返回码都必须靠调用者查阅和解释。这将是一个乏味并且费时的工作。这个策略的实现需要调用者在每一次调用的时候对返回值进行检查。每当一个错误码被检测到时，就会终止正常的执行流程并且把错误码传递给调用者。这些附加在每一个函数调用上的代码会使程序的大小急剧膨胀，并且引起软件维护和程序可读性的降低。

更糟糕的是，有时要想返回一个 `error value`(错误码)是不可能的。例如，构造函数没有返回值，所以就不能应用这种方法在对象构造失败的情况下报告错误。

第二种可以选择的用来报告运行期错误的途径是使用全局标记。它用来表明最后的操作是否成功。不像返回码策略，这个方法是标准化的。C 的 `<errno.h>` 头文件中定义了一种机制用来检查和给一个全局整型标记“`errno`”赋值。

这种策略也有不能被忽视的固有缺陷。例如，在一个多线程环境中，被一个线程赋予了一个错误码的 `errno` 有可能不经意地被另一个线程所改写，而调用者还未对 `errno` 进行检查。另外，对错误码而不是一个更为可读的信息进行使用是很不利的，因为那些错误码可能会在不同的环境中不兼容。最终，这种方法需要严格的且良好的编程样式，也就是不断地对 `errno` 的当前值进行检查。

全局标记策略和函数返回值策略是相似的：二者都提供一种机制来报告错误，但是二者却都不能保证错误被处理。例如，一个函数没有成功打开一个文件可以通过给 `errno` 赋予一个合适的值来表明错误的发生。然而，它不能阻止另一个函数试图写入和关闭那个文件。更进一步，如果 `errno` 表明一个错误并且程序员检测到而且按照预期处理了它，那么 `errno` 还应该被显式地复位。如果一个程序员忘记了做这件事，那么将会引起其他函数误以为错误还没有被处理，从而去校正那个问题，引起不可预知的结果。

最为残酷的处理运行期错误的方法是简单地终止程序。虽然这种解决方案去除了上面两种方法的一些缺点；例如，没有必要反复地检查每个函数返回值的状态，而且程序员也不必赋值给一个全局标记，反复地测试和清除它的值。

在标准 C 的函数库中有两个函数用来终止一个程序：`exit()`和 `abort()`函数。`exit()`被调用表明程序被成功终止，或者它可以在遇到运行期错误的时候被调用。在把控制权交还给运行环境之前，`exit()`首先会清空流和关闭打开的文件。`abort()`却不一样，它表示程序被意外终止，不会清空流和关闭打开的文件。

关键性的程序不应该在任何运行期存在错误的情况下突然终止。如果一个生命支持系统突然停止工作仅仅是因为它的控制器检测到 0 做除数，那么将是一种灾难。同样，一个控制由人驾驶的航天飞机自动运行的计算机系统也不应该因为暂时和地面控制系统失去联系就停止工作。类似的，电话公司的账目系统或者银行系统都不应该在运行期出现错误的时候就中止。健壮的真实世界的应用程序应该做得更好。

程序终止甚至对于任何应用程序都是不完美的。一个检测到错误的函数通常都没有必要的信息来衡量错误的严重性。例如一个内存分配函数并不能说出内存分配失败是由于用户正在使用调试器、网页浏览器、电子制表软件、文字处理软件，还是由于系统因为硬件错误变得不稳定。在第一种情况下，系统可以简单地显示一条信息来告诉用户关闭不必要的应用程序。在第二种情况下，就需要一种程序员自行处理内存回收、更为残酷的措施了。然而，在终止程序的策略下，那个内存分配函数就会简单地终止程序，而不考虑错误的严重性。这种方法在一些关键性应用程序中是无法应用的。好的系统设计应该保证运行期错误被检测和报告，但是它也应该确保最小限度的容错水平。

另外，`abort()`和 `exit()`函数却不可在面向对象环境中使用，因为 `exit()`和 `abort()`不销毁对象。对象可以持有从构造函数或者某个成员函数中获得的资源：从 free store 中分配的内存、文件句柄、通信端口和 I/O 设备等。这些资源必须在适当时候被释放。通常，资源都是由析构函数来释放。这种设计方法被称为资源初始化也就是获得资源(resource initialization is acquisition)。在栈上建立的局部对象会自动销毁。然而 `abort()`和 `exit()`并不调用这些局部对象的析构函数。因此，程序的意外终止将会引起无法挽回的损害：数据库被破坏，文件可能丢失，并且一些有价值的信息可能丢失。基于这个原因，设计者无法在面向对象环境中使用 `abort()`和 `exit()`。

18.2 异常处理的思想

传统 C 的错误处理方法并不适合 C++，C++ 的一个设计目标就是让用 C++ 进行大规模

软件开发比 C 更好更安全。

C++的设计者们已经意识到缺乏合适的错误处理机制会使得实现这一目标相当困难。他们试图寻找一种完全摆脱 C 的错误处理缺陷的解决方案。其中的一种想法就是建立在当异常被触发的时候程序自动把控制权传递给系统。这种机制必须简单,并且它能够使程序员从不断地检查一个全局标记或者返回值的苦差事中解脱出来。另外,它还必须保证异常处理程序能够自动获得异常信息。最终它还要确保当一个异常没有在本地处理的时候,本地对象能够被适当地销毁,并且把它所持有的资源释放。

小型程序在出现异常时,一般是将程序立即中断运行,无条件释放所有资源。

以下示例可实现当除数为零时,停止运行并给出提示信息。

```
#include<iostream.h>
#include<stdlib.h>
double fuc(double x, double y)
{
    if(y==0)
    {
        cerr<<"error of dividing zero.\n";
        exit(1);
    }
    return x/y;
}
void main()
{
    fuc(2,3);
    fuc(4,0);
}
```

在大中型程序中,上述处理方法就过于简单粗糙。这是因为在大中型程序中,函数之间有着明确的分工和复杂的调用关系。发现错误的程序往往在函数调用链的低层,这样,简单地在发现错误的函数中处理异常,就没有机会把调用链中的上层函数已经完成的一些工作做妥善的善后处理。例如,上层函数已经申请了堆对象,那么释放堆对象的工作显然不能在底层函数中处理,从而使程序不能正常运行。因此,对于大中型程序来说,在程序运行中一旦发生异常,应该允许恢复和继续运行。恢复是指把产生异常的错误处理掉,中间可能要涉及一系列函数调用链的退栈、对象的析构和资源的释放等。继续运行是指在异常处理之后,在紧接着异常处理的代码区域中继续运行。

处理异常的基本思想是:在底层发生的问题,逐级上报,直到有能力可以处理异常的那级为止。即在应用程序中,若某个函数发现了错误并引发异常,这个函数就将该异常向上级调用者传递,请求调用者捕获该异常并处理该错误。如果调用者不能处理该错误,就

继续向上级调用者传递，直到异常被捕获错误被处理为止。如果程序最终没有相应的代码处理该异常，那么该异常最后被 C++ 系统所接受，C++ 系统就简单地终止程序运行。异常的传递如图 18-1 所示。

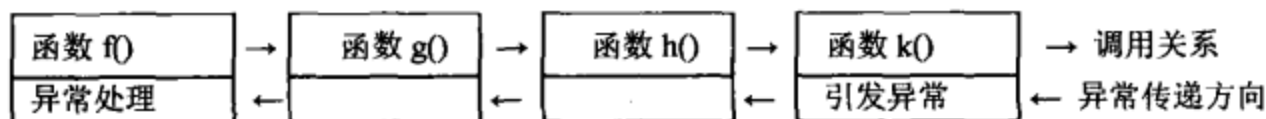


图 18-1 异常的传递

从图中可见，C++ 异常处理的目的，是在异常发生时，尽可能地减少破坏，周密地处理善后，而不去影响程序其他部分的运行。

18.3 异常处理的实现方式

C++ 异常处理的步骤如下。

- (1) 定义异常(try 语句块)，将可能产生错误的语句放在 try 语句块中。其格式是：

```
try
{
    可能产生错误的语句
}
```

- (2) 定义异常处理(catch 语句块)，将异常处理的语句放在 catch 语句块中，以便异常被传递来时处理。通常，异常处理是放在 try 语句块后的由若干个 catch 语句组成的程序，其格式是：

```
catch(异常类型声明 1)
{
    异常处理语句块 1
}
catch(异常类型声明 2)
{
    异常处理语句块 2
}
...
catch(异常类型声明 n)
{
    异常处理语句块 n
}
```

- (3) 抛出异常(throw 语句)，检测是否产生异常，若是，则抛出异常。其格式是：

throw 表达式;

例如: `throw ExceptionClass("It's a exception!");`

例句中, `ExceptionClass` 是一个类, 它的构造函数以一个字符串作为参数, 用来说明异常。也就是说, 在抛出异常的时候, C++的编译器先构造一个 `ExceptionClass` 的对象, 让它作为 `throw` 的返回值抛出去。同时, 程序返回, 调用析构。看下面这个例子。

```
class ExceptionClass
{
    char* name;
public:
    ExceptionClass(char* name="default name")
    {
        cout<<"Construct " <<name<<endl;
        this->name=name;
    }
    ~ExceptionClass()
    {
        cout<<"Destruct " <<name<<endl;
    }
    void mythrow()
    {
        throw ExceptionClass("o,my god");
    }
};
void main()
{
    ExceptionClass e("haha");
    try
    {
        e.mythrow();
    }
    catch(...) {}
}
```

程序执行 `e.mythrow()` 函数, 如果运行时遇到异常, 则从 `catch` 处开始执行, 否则无视 `catch` 的存在。

18.4 构造和析构中的异常抛出

先看个程序, 假如在构造函数的地方抛出异常, 这个类的析构会被调用吗? 可如果不调用, 那类里的东西岂不是不能被释放了? 例如:

```

#include <iostream.h>
#include <stdlib.h>

class ExceptionClass1{
    char* s;
public:
    ExceptionClass1(){
        cout<<"ExceptionClass1()"<<endl;
        s=new char[4];
        cout<<"throw a exception"<<endl;
        throw 18;
    }
    ~ExceptionClass1(){
        cout<<"~ExceptionClass1()"<<endl;
        delete[] s;
    }
};

void main(){
    try{
        ExceptionClass1 e;
    }catch(...)
    {}
}

```

程序输出结果如下：

```

ExceptionClass1()
throw a exception

```

运行结果到此为止了！可是，在这两句输出之间，程序已经给 s 分配了内存，哪里去了？内存释放了吗？没有，因为它是在析构函数中释放的。

为了避免这种情况，应避免对象通过本身的构造函数涉及异常抛出，即既不在构造函数中出现异常抛出，也不应在构造函数调用的一切东西中出现异常抛出。

那么，在析构函数中的情况呢？现在已经知道的是，异常抛出之后，就要调用本身的析构函数，如果这析构函数中还有异常抛出的话，则已存在的异常尚未被捕获，会导致异常捕捉不到。也就是说，尽量不要在构造函数和析构函数中存在异常抛出。

18.5 异常规格说明

如果程序调用其他的函数，里面有异常抛出，需要去查看它的源代码去看看都有什么

异常抛出吗？可以，但是太麻烦。比较好的解决办法是，在编写带有异常抛出的函数时，采用异常规格说明，使当前程序看到函数声明就知道有哪些异常出现。异常规格说明大体上为以下格式：

```
void ExceptionFunction(argument...) throw(ExceptionClass1, ExceptionClass2, ...)
```

所有异常类都在函数末尾的 `throw()` 的括号中加以说明。这样，对于函数调用者来说，函数会抛出什么样的异常就十分清楚了。例如：

```
void f() throw(ExClass)
{
    ExClass exception;
    // ...
    throw(exception);
}
```

而一个空的异常规格声明表明函数不抛出任何异常，如：

```
void f() throw()
{
    // ... 函数不抛出异常...
}
```

如果函数没有异常规格声明，它可以抛出任何类型的异常，如：

```
void f()
{
    // ... 函数可以抛出任意异常 ...
}
```

异常声明是函数接口的一部分，它必须在头文件中的函数声明上指定。异常规范是函数和程序剩下部分之间的协议，它保证该函数不会抛出任何没有出现在其异常规范中的异常。如果函数声明指定了一个异常规范，则同一函数的重复声明必须指定同一类型的异常规范，而且同一函数的不同声明上的异常规范是不能累积的，例如：

```
// 同一函数的两个声明
extern int foo( int = 0 ) throw(string);
extern int foo( int parm ) { }           // 错误：异常规范被省略
```

如果函数抛出了一个没有被列在异常规范中的异常会怎么样？程序只有在遇到某种不正常情况时，异常才会被抛出。在编译时刻，编译器不可能知道在执行时程序是否会遇到这些异常，因此一个函数异常规范的违例只能在运行时刻才能被检测出来。如果函数抛出了一个没有被列在其异常规范中的异常，则系统会调用 C++ 标准库中定义的函数 `unexpected()`。而 `unexpected()` 的缺省行为则是调用 `terminate()` 来结束程序。

由于函数异常规范的违例只有在运行时刻才能被检测到，所以如果一个表达式能够抛

出一个不被规范允许的异常类型，则编译器不会产生编译时刻错误。并且，如果这个表达式不会被执行，或者它没有抛出违反异常规范的那个异常，则该程序会像期望的那样运行，而且该函数异常规范不会被发现。例如：

```
extern void doit( int, int ) throw(string, exceptionType);
void action ( int op1, int op2) throw(string)
{
    doit( op1, op2 );           // 没有编译错误
    // ...
}
```

函数 `doit()` 可以抛出一个 `exceptionType` 类型的异常，它不是函数 `action()` 的异常规范所允许的，但即使函数 `action()` 不允许这种类型的异常，该函数也能编译成功。编译器产生相应的代码以确保当违反异常规范的异常被抛出时调用运行库函数 `unexpected()`。

18.6 标准异常

正像许多人想象的一样，C++标准库有自己的标准异常类。在 C++标准库中，有些函数抛出特定的异常，而另外一些根本不抛出任何异常。因为 C++标准中没有明确规定，所以 C++的库函数可以抛出任何对象或不抛出。但 C++标准推荐运行库的实现通过抛出定义在 `<stdexcept>` 中的异常类型或其派生类型来报告错误，定义如下：

一个基类 `exception` 是所有 C++标准异常的基类：

```
class exception
{
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

在 `exception` 下面派生了以下两个异常类。

- `logic_error`：用于报告程序的逻辑错误，可在程序执行前被检测到。
- `runtime_error`：用于报告程序运行时的错误，只有在运行的时候才能检测到。

这两个类又分别有自己的派生类：

```
namespace std
{
```

```
class logic_error;           // : public exception
    class domain_error;      // : public logic_error
    class invalid_argument;  // : public logic_error
    class length_error;      // : public logic_error
    class out_of_range;      // : public logic_error
class runtime_error;         // : public exception
    class range_error;       // : public runtime_error
    class overflow_error;    // : public runtime_error
    class underflow_error;   // : public runtime_error
}
```

由 `logic_error` 派生的异常类有以下几个。

- `domain_error`: 报告违反了前置条件。
- `invalid_argument`: 指出函数的一个无效参数。
- `length_error`: 指出有一个产生超过 NPOS 长度的对象的企图(NPOS 为 `size_t` 最大的可表现值)。
- `out_of_range`: 报告参数越界。
- `bad_cast`: 在运行时类型识别中有一个无效的 `dynamic_cast` 表达式。
- `bad_typeid`: 报告在表达式 `typeid(*p)` 中有一个空指针 `p`。

由 `runtime_error` 派生的异常有以下几个。

- `range_error`: 报告违反了后置条件。
- `overflow_error`: 报告一个算术溢出。
- `bad_alloc`: 报告一个存储分配错误。

本章小结

本章讲述了 C++ 异常处理的机制, 通过几个简单的例子展示了异常处理的常用手段及类型, 其中心思想是提高代码的可读性。



附录 A 程序的写作风格

风格的作用主要就是使代码容易读，无论是对程序员本人，还是对其他人。好的风格对于好的程序设计具有关键性作用。我们希望最先谈论风格，也是为了使读者在阅读本书其余部分时能特别注意这个问题。

写好一个程序，当然需要使它符合语法规则、修正其中的错误和使它运行得足够快，但是实际应该做的远比这多得多。程序不仅需要给计算机读，也要给程序员读。一个写得好的程序比那些写得差的程序更容易读、更容易修改。经过了如何写好程序的训练，生产的代码更可能是正确的。幸运的是，这种训练并不太困难。

A.1 工程的概念

1968 年秋季，NATO(北约)的科技委员会召集了近 50 名一流的编程人员、计算机科学家和工业界巨头，讨论和制定摆脱“软件危机”的对策。在那次会议上第一次提出了软件工程(Software Engineering)这个概念。到今年(2010 年)，软件工程整整走过了 40 多年的历程。

在这 40 多年的发展中，人们针对软件危机的表现和原因，经过不断地实践和总结，越来越清楚地认识到按照工程化的原则和方法组织软件开发工作，是摆脱软件危机的一个主要出路。

今天，尽管“软件危机”并未被彻底解决，但软件工程 40 多年的发展仍可以说是硕果累累。

下面给出一个软件工程的定义，然后简单讨论一下软件工程所包括的内容。

软件工程是一门研究如何用系统化、规范化和数量化等工程原则和方法去进行软件的开发和维护的学科。

软件工程包括两方面内容：软件开发技术和软件项目管理。软件开发技术包括软件开发方法学、软件工具和软件工程环境；软件项目管理包括软件度量、项目估算、进度控制、人员组织、配置管理和项目计划等。

统计数据表明，大多数软件开发项目的失败，并不是由于软件开发技术方面的原因，

而是由于不适当的管理造成的。遗憾的是, 尽管人们对软件项目管理重要性的认识有所提高, 但在软件管理方面的进步远比在设计方法学和实现方法学上的进步小, 至今还提不出一套管理软件开发的通用指导原则。

对于软件工程的研究目前已经成为一门独立的学科, 在本书中并不打算过多地讨论这方面的内容, 而是从编程风格的角度给大家介绍大规模软件开发中一些常识性的知识, 以便为大家后续课程的学习打下基础。

A.2 命名规则与命名空间

1. 匈牙利命名法及其前缀

比较著名的命名规则当推 Microsoft 公司的“匈牙利命名法”, 该命名规则的主要思想是“在变量和函数名中加入前缀以增进人们对程序的理解”。匈牙利命名法为 C 标识符的命名定义了一种非常标准化的方式, 这种命名方式是以以下两条规则为基础的。

(1) 标识符的名字以一个或者多个小写字母开头, 用这些字母来指定数据类型。表 A-1 列出了常用数据类型的标准前缀。

表 A-1 常用数据类型的标准前缀

前 缀	类 型
a	数组 (Array)
b	布尔值 (Boolean)
by	字节 (Byte)
c	有符号字符 (Char)
cb	无符号字符 (Char Byte, 没有多少人用)
cr	颜色参考值 (ColorRef)
cx, cy	坐标差(长度 ShortInt)
dw	双字(Double Word)
fn	函数
h	句柄(Handle)
i	整型
l	长整型(Long Int)
lp	长整型指针(Long Pointer)
m	类的成员

续表

前 缀	类 型
n	短整型 (Short Int)
np	近指针(Near Pointer)
p	指针(Pointer)
s	字符串型
sz	以 null 做结尾的字符串型 (String with Zero End)
w	字

(2) 在标识符内, 前缀以后就是一个或者多个第一个字母大写的单词, 这些单词清楚地指出了源代码内那个对象的用途。比如, `m_szStudentName` 表示一个学生名字类成员变量, 数据类型是字符串型。

“匈牙利命名法”最大的缺点是繁琐, 例如:

```
int i, j, k;
float x, y, z;
```

倘若采用“匈牙利”命名规则, 则应当写成:

```
int iI, iJ, iK;      // 前缀 i 表示 int 类型
float fX, fY, fZ;    // 前缀 f 表示 float 类型
```

如此繁琐的程序会让绝大多数程序员无法忍受。

2. 关于命名的共性规则

实际上没有一种命名规则可以让所有的程序员赞同, 程序设计教科书一般都不指定命名规则。命名规则对软件产品而言并不是“成败攸关”的事, 因此不必要花费太多精力试图发明世界上最好的命名规则, 而应当制定一种令大多数项目成员满意的命名规则, 并在项目中贯彻实施。

本节论述的共性规则是被大多数程序员采纳的, 实际工作中应当在遵循这些共性规则的前提下, 再扩充特定的规则。

(1) 标识符应当直观且可以拼读, 可望文知意, 不必进行“解码”。

标识符最好采用英文单词或其组合, 便于记忆和阅读。切忌使用汉语拼音来命名。程序中的英文单词一般不会太复杂, 用词应当准确。例如不要把 `CurrentValue` 写成 `NowValue`。

(2) 标识符的长度应当符合“min-length && max-information”原则。

几十年前老 ANSIC 规定名字不准超过 6 个字符, 现今的 C++/C 不再有此限制。一般来说, 长名字能更好地表达含义, 所以函数名、变量名、类名长达十几个字符也不足为怪。

那么名字是否越长越好？不见得！例如变量名 `maxVal` 就比 `maxValueUntilOverflow` 好用。单字符的名字也是有用的，常见的如 `i`、`j`、`k`、`m`、`n`、`x`、`y` 和 `z` 等，它们通常可用作函数内的局部变量。

(3) 命名规则尽量与所采用的操作系统或开发工具的风格保持一致。

例如 Windows 应用程序的标识符通常采用“大小写”混排的方式，如 `AddChild`；而 Unix 应用程序的标识符通常采用“小写加下划线”的方式，如 `add_child`。别把这两类风格混在一起用。

(4) 程序中不要出现仅靠大小写区分的相似的标识符。

例如：

```
int x, X;           // 变量 x 与 X 容易混淆
void foo(int x);    // 函数 foo 与 FOO 容易混淆
void FOO(float x);
```

(5) 程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解。

(6) 变量的名字应当使用“名词”或者“形容词+名词”。

例如：

```
float fValue;
float fOldValue;
float fNewValue;
```

(7) 用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

例如：

```
int nMinValue;
int nMaxValue;
int SetValue(...);
int GetValue(...);
```

(8) 尽量避免名字中出现数字编号，如 `nValue1`、`nValue2` 等，除非逻辑上的确需要编号。这是为了防止程序员偷懒，不肯为命名动脑筋而导致产生无意义的名字(因为用数字编号最省事)。

A.3 注释的书写规范

C++语言中，程序块的注释常采用“`/*...*/`”，行注释一般采用“`//...`”。注释通常用于以下几种情况。

- 版本、版权声明。
- 函数接口说明。
- 重要的代码行或段落提示。

注释应描述正在发生什么事、如何完成它、参数表示什么、使用了哪些全局变量以及任何限制或错误。但要避免不必要的注释。如果代码比较清晰，并且使用了良好的变量名，那么它应该能够较好地说明自身。

下面是一种多余的注释风格：

```
i=i+1;          /* Add one to i */
```

很明显变量*i*递增了1。还有更糟糕的注释方法：

```
/* Add one to i          */
i=i+1;
```

因为编译器不检查注释，所以不保证它们是正确的。在写程序时应边写代码边注释，修改代码的同时应修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。与代码不一致的注释会起到相反的作用。过多的注释会使代码混乱。

尽量避免在注释中使用缩写，特别是不常用缩写。注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。程序的注释如图 A-1 所示。

<pre>/* * 函数介绍: * 输入参数: * 输出参数: * 返回值 : */ void Function(float x, float y, float z) { ... }</pre>	<pre>if (...) { ... while (...) { ... } // end of while ... } // end of if</pre>
--	--

图 A-1 程序的注释示例

A.4 程序的版式

版式虽然不会影响程序的功能，但会影响可读性。程序的版式追求清晰、美观，是程序风格的重要构成因素。

可以把程序的版式比喻为“书法”。好的“书法”可让人对程序一目了然，看得兴致勃

勃。差的“书法”如螃蟹爬行，让人看得索然无味，更令维护者烦恼有加。

1. 空行

空行起着分隔程序段落的作用。空行得体(不过多也不过少)将使程序的布局更加清晰。空行不会浪费内存，虽然打印含有空行的程序是会多消耗一些纸张，但是值得。所以不要舍不得用空行。有以下两个建议。

- 在每个类声明之后、每个函数定义结束之后都要加空行，参见图 A-2(a)。
- 在一个函数体内，逻辑上密切相关的语句之间不加空行，其他地方应加空行分隔。参见图 A-2(b)。

<pre>// 空行 void Function1(...) { ... } // 空行 void Function2(...) { ... } // 空行 void Function3(...) { ... }</pre>	<pre>// 空行 while (condition) { statement1; // 空行 if (condition) { ... } else { ... } // 空行 statement2; }</pre>
(a) 函数之间的空行	(b) 函数内部的空行

图 A-2 空行示例

2. 代码行

对于代码行有以下几个建议。

- 一行代码只做一件事情，只写一条语句。这样的代码容易阅读，并且方便于写注释。
- if、for、while、do 等语句自占一行，执行语句不得紧跟其后。
- 建议尽可能在定义变量的同时初始化该变量(就近原则)。

如果变量的引用处和其定义处相隔比较远，变量的初始化很容易被忘记。如果引用了未被初始化的变量，可能会导致程序错误。本建议可以减少隐患。例如：

```
int width = 10;      // 定义并初始化 width
int height = 10;     // 定义并初始化 height
int depth = 10;      // 定义并初始化 depth
```


- 如果 if、for、while、do 等语句后的执行语句只有一行，则可以不加 { }。

3. 代码行内的空格

对代码行内的空格有以下几个建议。

(1) 关键字之后要留空格。像 const、virtual、inline、case 等关键字之后至少要留一个空格，否则无法辨析关键字。像 if、for、while 等关键字之后应留一个空格再跟左括号 ‘(’，以突出关键字。

(2) 函数名之后不要留空格，紧跟左括号 ‘(’，以与关键字区别。

(3) ‘(’ 向后紧跟，‘)’、‘,’、‘;’ 向前紧跟，紧跟处不留空格。

(4) ‘,’ 之后要留空格，如 Function(x, y, z)。如果 ‘;’ 不是一行的结束符号，其后要留空格，如 for (initialization; condition; update)。

(5) 赋值操作符、比较操作符、算术操作符、逻辑操作符和位域操作符，如 “=”、“+=”、“>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”和“^”等二元操作符的前后应当加空格。

(6) 一元操作符如 “!”、“~”、“++”、“--”和“&”(地址运算符)等前后不加空格。

(7) 像 “[]”、“.”、“->”这类操作符前后不加空格。

(8) 对于表达式比较长的 for 语句和 if 语句，为了紧凑起见可以适当地去掉一些空格。如 for (i=0; i<10; i++)和 if ((a<=b) && (c<=d))。

具体例子如图 A-3 所示。

void Funcl(int x, int y, int z);	// 良好的风格
void Funcl (int x,int y,int z);	// 不良的风格
if (year >= 2000)	// 良好的风格
if(year>=2000)	// 不良的风格
if ((a>=b) && (c<=d))	// 良好的风格
if(a>=b&& c<=d)	// 不良的风格
for (i=0; i<10; i++)	// 良好的风格
for(i=0;i<10;i++)	// 不良的风格
for (i = 0; I < 10; i ++)	// 过多的空格
x = a < b ? a : b;	// 良好的风格
x=a<b?a:b;	// 不好的风格
int *x = &y;	// 良好的风格
int * x = & y;	// 不良的风格
array[5] = 0;	// 不要写成 array [5] = 0;
a.Function();	// 不要写成 a . Function();
b->Function();	// 不要写成 b -> Function();

图 A-3 代码行内的空格

4. 对齐

- 程序的分界符 ‘{’ 和 ‘}’ 应独占一行并且位于同一列，同时与引用它们的语句左对齐。
- {} 之中的代码块在 ‘{’ 右边数格处左对齐。

图 A-4(a)所示为风格良好的对齐，图 A-4(b)所示为风格不良的对齐。

<pre>void Function(int x) { ... // program code }</pre>	<pre>void Function(int x){ ... // program code }</pre>
<pre>if (condition) { ... // program code } else { ... // program code }</pre>	<pre>if (condition){ ... // program code } else { ... // program code }</pre>
<pre>For (initialization; condition; update) { ... // program code }</pre>	<pre>for (initialization; condition; update){ ... // program code }</pre>
<pre>while (condition) { ... // program code }</pre>	<pre>while (condition){ ... // program code }</pre>
<p>如果出现嵌套的 {}，则使用缩进对齐，如：</p> <pre>{ ... { ... } ... }</pre>	

(a) 风格良好的对齐

(b) 风格不良的对齐

图 A-4 对齐示例

5. 长行拆分

代码行最大长度宜控制在 70~80 个字符以内。代码行不要过长，否则眼睛看不过来，也不便于打印。

长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首(以便突出操作符)。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

长行拆分的例子如图 A-5 所示。

<pre> if ((very_longer_variable1 >= very_longer_variable12) && (very_longer_variable3 <= very_longer_variable14) && (very_longer_variable5 <= very_longer_variable16)) { dosomething(); } </pre>
<pre> virtual CMatrix CMultiplyMatrix (CMatrix leftMatrix, CMatrix rightMatrix); </pre>
<pre> for (very_longer_initialization; very_longer_condition; very_longer_update) { dosomething(); } </pre>

图 A-5 长行的拆分示例

6. 修饰符的位置

修饰符 “*” 和 “&” 应该靠近数据类型还是变量名，是个有争议的话题。

若将修饰符 * 靠近数据类型，例如：int* x; 从语义上讲此写法比较直观，即 x 是 int 类型的指针。

上述写法的弊端是容易引起误解，例如：int* x, y;，此处 y 容易被误解为指针变量。虽然将 x 和 y 分行定义可以避免误解，但并不是人人都愿意这样做。

下面是笔者的两个建议。

(1) 在变量定义时，应当将修饰符 * 和 & 紧靠变量名。

例如：

```

char *name;
int *x, y;      // 此处 y 不会被误解为指针

```

(2) 作为函数的返回值，修饰符 * 和 & 应当紧靠数据类型。

例如：

```

CTest* Function();

```

7. 类的版式

类可以将数据和函数封装在一起，其中函数表示了类的行为(或称服务)。类提供关键

字 `public`、`protected` 和 `private`，分别用于声明哪些数据和函数是公有的、受保护的或者是私有的。这样可以达到信息隐藏的目的，即让类仅仅公开必须要让外界知道的内容，而隐藏其他一切内容。不可以滥用类的封装功能，不要把它当成火锅，什么东西都往里扔。

类的版式主要有以下两种方式。

(1) 将 `private` 类型的数据写在前面，而将 `public` 类型的函数写在后面，如图 A-6(a)所示。采用这种版式的程序员主张类的设计“以数据为中心”，重点关注类的内部结构。

(2) 将 `public` 类型的函数写在前面，而将 `private` 类型的数据写在后面，如图 A-6(b)所示。采用这种版式的程序员主张类的设计“以行为为中心”，重点关注的是类应该提供什么样的接口(或服务)。

很多 C++ 教科书受到 Bjarne Stroustrup 第一本著作的影响，不知不觉地采用了“以数据为中心”的书写方式，但并不见得有多少道理。

我建议读者采用“以行为为中心”的书写方式，即首先考虑类应该提供什么样的函数。这是很多人的经验——“这样做不仅让自己在设计类时思路清晰，而且方便别人阅读。因为用户最关心的是接口，谁愿意先看到一堆私有数据成员！”

<pre> class A { private: int i, j; float x, y; ... public: void Func1(void); void Func2(void); ... } </pre>	<pre> class A { public: void Func1(void); void Func2(void); ... private: int i, j; float x, y; ... } </pre>
---	---

(a) 以数据为中心的版式

(b) 以行为为中心的版式

图 A-6 类的版式

A.5 程序版本控制

A.5.1 版本控制的方法及目的

版本控制系统(Version Control System, VCS)用来记录源文件的历史信息，甚至二进制文件、媒体文件等。

当软件修改时有时会产生 Bugs，并且你可能在做这次修改后很长时间不会发现这些 Bugs。引入版本控制，你可以容易地回顾老的代码版本去发现哪一次的修改导致这些问题。这样做有时候会非常有帮助。

你可能会保留你每一次的代码版本，这是很多程序员在最初学写程序时最爱做的事情，但这样会浪费你很多的代码空间。版本控制却可以保存你的程序的多个版本在一个文件中。它仅仅保留版本间的不同内容。

版本控制可以协助一个团队共同开发一个工程。如果你是一个项目中的一组成员之一，除非你特别仔细，你很容易覆盖其他人的工作。一些编辑器，例如 GNU Emacs，试图去判定一个文件是否被两人同时修改。遗憾的是，如果一个人使用其他的编辑器，这个安全方式将不再有效。VCS 让不同开发者独立工作的方式解决了这个问题。每一个开发者的工作都在他自己的目录内，并且 VCS 将在每个开发者的工作完成后进行合并工作。

这样，通过在开发过程中实施版本控制，便可以做以下的事。

- 将整个软件版本恢复到以前某一时间的状态。
- 控制某一程序在同一时间只能由一个开发人员修改。
- 限制随意修改程序。
- 对每个开发人员编写的程序质量进行评估。

A.5.2 在软件开发过程中引入版本控制软件

1. 随时将程序恢复到以前某一时间点

版本控制软件可以将某一程序恢复到以前某一时间的状态，甚至将整个软件版本恢复到以前某一时间的状态，以保证程序在修改动荡后可以随时返回某一稳定历史时期的版本。

2. 实现程序的互斥性修改

要求实现某一程序在同一时间只能由一个开发人员修改。其具体实现方式是：需要修改程序的开发人员从源文件存放处提出(Check-out)一个程序，这时其他开发人员就不可以再 Check-out 同一个程序了，只有当第一个开发人员修改测试完成后，将更新版本的代码做放入(Check-in)操作，其他开发人员才能 Check-out 同一个程序。

3. 对程序修改进行有效的管理

在版本控制软件中可以将用户分为管理员和程序员两种角色，只有管理员可以将程序冻结(Freeze)和解冻(Unfreeze)，被冻结的程序是不允许修改的。修改程序的流程如下。

(1) 用户提交需求书, 程序员提交程序设计说明书, 项目主管审核通过后, 管理员将程序解冻。

(2) 由程序员 Check-out 程序。

(3) 程序员修改程序。

(4) 修改完成后程序员提交测试请求给测试小组, 测试小组进行测试; 如果测试不通过, 转向第(3)步。

(5) 测试通过以后程序员填写本次修改解释, 然后 Check-in 程序。

(6) 管理员将程序冻结。

至此完成一次程序的修改。在软件开发后期或者软件正式投入使用时, 这种方式对保证软件的稳定运行能起到非常重要的作用。

4. 将开发环境与测试环境、运行环境进行有效的隔离

测试环境与开发环境、运行环境分隔, 使用不同的物理机器环境。

5. 评估软件开发人员编写的程序质量, 控制软件开发的进度

使用详细的文件修改日志文件, 记录各程序员对程序的修改内容描述及文件清单。

6. 管理文档

由系统分析员定期对程序文档进行过滤分类, 整理成管理文档, 描述系统各部分程序的详尽实现功能及相互关联。

A.5.3 具体实施

1. 建立用户

对建立用户应注意以下两个要点。

(1) 首先在 NT 环境下建立程序开发小组专用的目录管理程序文档。初始环境需注意清除所有垃圾文件。

(2) 小心建立各用户对此目录的操作权限。

2. 规范程序员操作

为规范程序员操作, 应注意以下几个要点。

(1) 避免直接对生产机的直接操作, 规范流程为: 开发机开发→测试机测试→生产机运行。

(2) 因开发人员使用的开发工具的多样化情况, 所以提倡尽量规范开发工具, 尽量使

用有感知的编辑工具，如：Dreamweaver、UltraEdit 等。

(3) 在完成开发及初步测试后，提交相应开发文档至规定路径并通知系统分析员。

3. 将程序登记入库

新程序在测试机正常运行成功无误后，由系统管理员上传更新生产机，并记录上传日志(可简化为 Task Item，项目标识与开发文档对应)。

A.5.4 版本控制软件的使用

Microsoft Visual SourceSafe 6.0 是微软公司开发的版本控制软件，用于软件开发过程中的过程及版本管理，具有管理方便、使用简单的特点，很适合团队开发中的过程及版本控制。其使用步骤描述如下。

1. 安装

在使用 Microsoft Visual SourceSafe 6.0 的过程中，通常是采用一台机器作为服务器机，用于对软件的统一、集中存放管理；其他客户端机器连接该服务器上的相应数据库，以实现客户端机器之间版本的一致(或者使用一台客户端机器作为服务器，这里服务器和客户端机器只是概念上的称呼，以使用较高性能的机器作为服务器机为宜)。

1) 服务器端安装

安装 Microsoft Visual Studio 6.0 中的 Visual SourceSafe 6.0 软件，将其安装在指定的路径下，通常为 C:\Program Files\Microsoft Visual Studio(用户可以根据自己的情况做适当修改)，Visual Source Safe 6.0 服务器端软件位于其下面的 Common\vss 目录下，其他目录为该软件相应的附加文件所在的目录。安装完成后，退出。

2) 客户端安装

客户端的安装一般在服务器安装之后(因为客户端的安装要用到服务器的文件及资源)。客户端的安装文件位于已安装到服务器端的 C:\Program Files\Microsoft Visual Studio\VSS 目录下的可执行文件 netsetup.exe 中，由于要在其他客户机上安装，因此需要将 netsetup.exe 文件所在的目录共享。执行该文件，完成客户端的软件安装。

2. 配置

1) 服务器端配置

在服务器端机器上需要配置数据库及用户，以建立相应的控制管理体系。一般对于一个项目(或者工程)都要建立一个数据库，并且建立对应该数据库的若干使用用户(以用于分级、权限的管理)。

(1) 数据库的配置。

在服务器端单击安装后的“程序”→Microsoft Visual Studio 6.0→Microsoft Visual SourceSafe→Visual SourceSafe 6.0 Admin 命令，打开如图 A-7 所示的管理界面。

单击菜单 Tools→Create Database...命令，选择数据库所要存放的路径(建议单独为该数据库建立一个目录，以专门使用)，假定为“D:\VSSTest”。选择完成后，单击 OK 按钮即建立对应于这个目录的数据库“VSSTest”，完成后系统将会给出相应的提示信息。

单击菜单 Users→Open SourceSafe Database...命令，打开数据库选择对话框；单击 Browse...按钮，选择刚建立的数据库“VSSTest”目录下的配置文件“srcsafe.ini”，单击“打开”按钮，则该目录名出现在可用的数据库列表名称中，如图 A-8 所示。

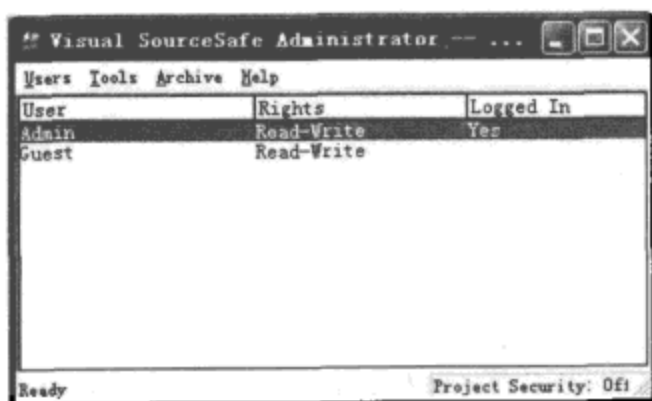


图 A-7 Visual SourceSafe 6.0 管理界面

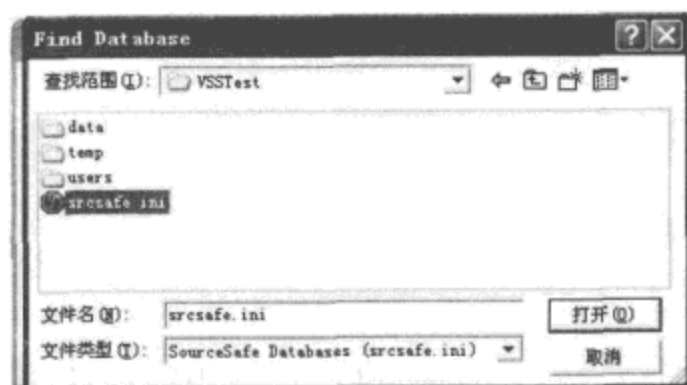


图 A-8 srcsafe.ini 文件的存储位置

此时单击数据库选择对话框右边的 Open 按钮，系统会提示管理员更改超级用户“Admin”的密码，单击 OK 按钮即可。打开管理界面，建议立即修改超级用户“Admin”的密码：选择用户 Admin，单击菜单 Users→Change Password...命令，输入新的密码，并确认(旧密码无须输入，因为默认为空)，单击 OK 按钮保存。之后，建议更改“Guest”用户的权限，使其只有只读权限(因为默认情况下，该用户拥有同“Admin”用户相同的权限)，操作如下：双击“Guest”用户，在弹出的对话框中选择其中的 Read Only 复选框，然后单击 OK 按钮即可，如图 A-9 所示。

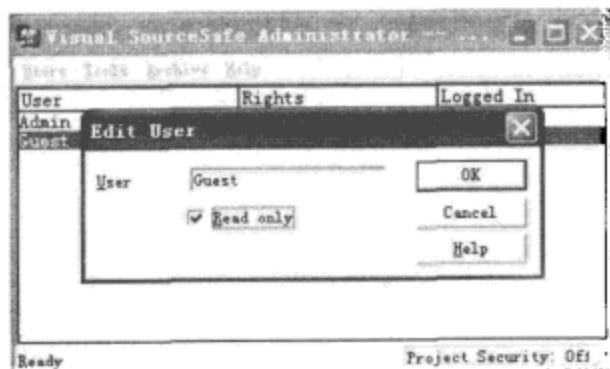


图 A-9 更改“Cuest”用户的权限

(2) 项目安全的设置。

打开管理界面之后,需要打开用户的安全设置功能(系统默认是关闭的)。单击菜单 Tools → Options...命令,打开 SourceSafe Options 对话框,单击 Project Security 标签页,此时可以看到 Enable Project Security 选项是未选中的,其下面的 Default User Rights 也是灰色的。我们选择 Enable Project Security 选项,并对下面的默认用户权限仅选择 Read 复选框,如图 A-10 所示,单击“确定”按钮以保存修改。之后,在建立用户后,新建用户的默认权限为“Read”,可根据项目中成员的角色情况对工程文件中不同的目录进行详细的权限设置(见“(4)>用户及权限的设置”)。



图 A-10 设置新建用户的默认权限

(3) 项目文件的导入。

通常在建立了数据库之后,其中还没有项目文件,此时需要将要管理的项目下的文件导入该数据库中,其步骤如下。

① 在 VC.net 中新建一个项目取名为“VSSexample”,如果已经有项目可以直接打开,如图 A-11 所示。

② 将项目加入 SourceSafe。

在 Solution Explorer(解决方案资源管理器)中,右击“解决方案”节点,然后单击“将解决方案添加到源代码管理”命令,如图 A-12 所示。按照源代码管理提供程序的要求提供数据库位置(要把这个项目添加到哪个数据库)和用户登录信息,如图 A-13 所示。把这个项目添加到上面建的那个数据库中,如图 A-14 所示。

选完数据库后,会得到提示项目在数据库中的存储结构,先是提示这个方案的解决方案在数据库中的存储位置(解决方案是比项目高一级的单位,一个解决方案可以包括多个项

目，在.net 中任何项目都必须包括在一个解决方案中，若没指定解决方案也会给你指定一个跟这个项目同名的解决方案，比如项目名是 VSSexample，vs.net 自动给添加了一个叫 VSSexample 的解决方案)，也可以重命名为其他方案名，在 Project 文本框中填入解决方案名 VSSexample，单击 Create 按钮，会在数据库的根目录下建立一个 VSSexample 目录，如图 A-15 所示。

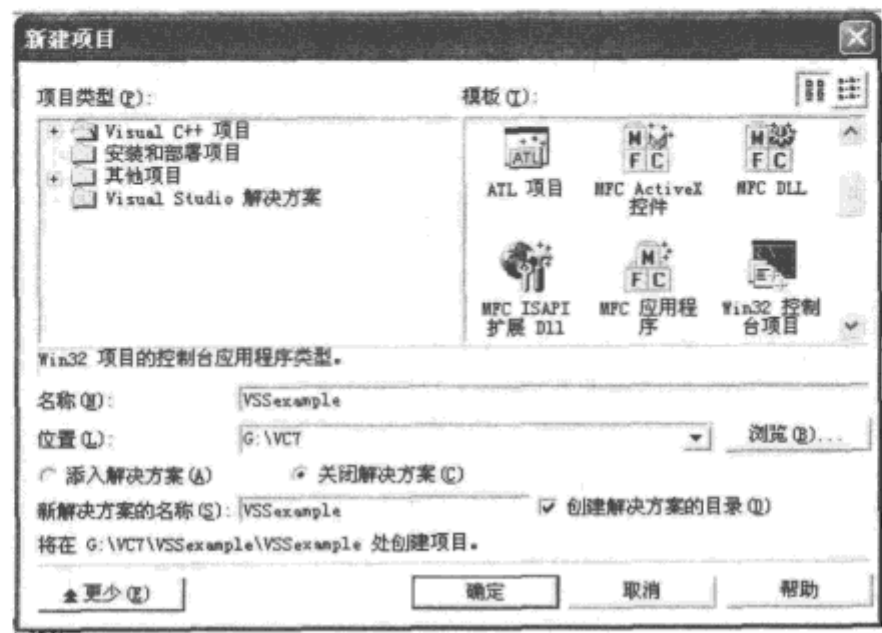


图 A-11 新建项目

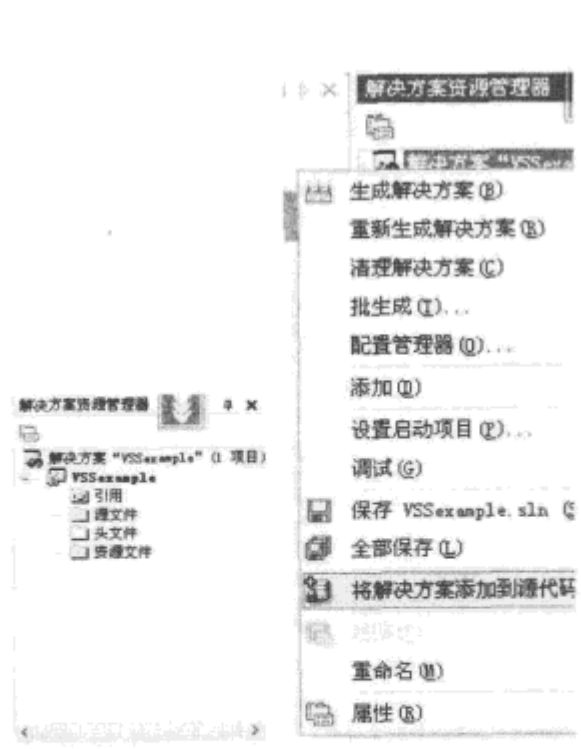


图 A-12 添加解决方案



图 A-13 选择数据库并登录

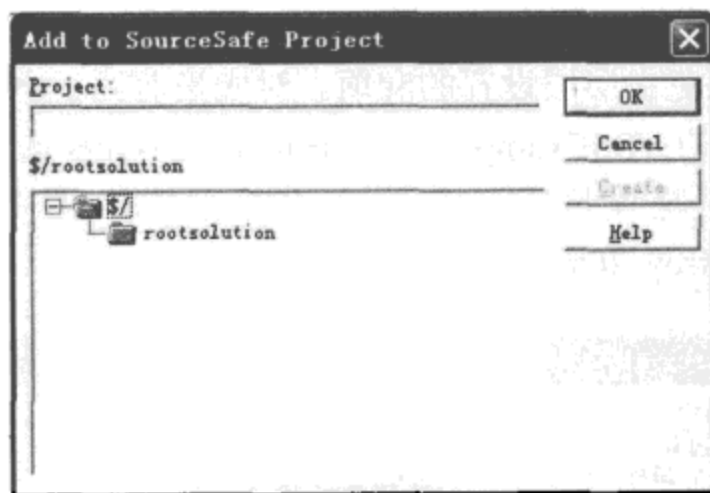


图 A-14 添加项目(1)

在根解决方案目录下，输入项目文件夹的名称，该文件夹将包含项目文件的主控副本。

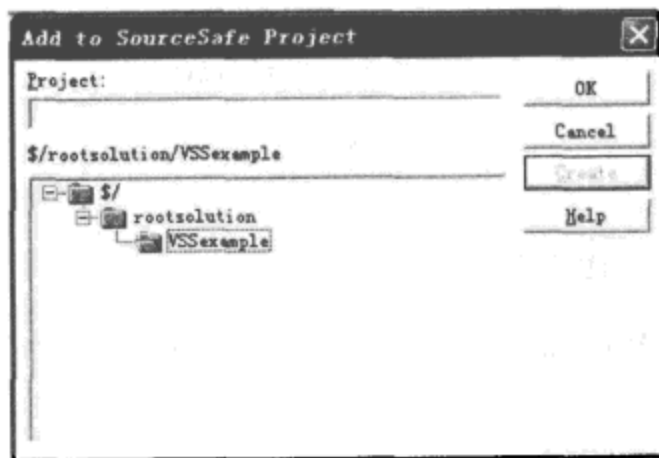


图 A-15 添加项目(2)

此时，项目工程中的文件便已经导入 VSS 数据库，并纳入版本控制之中；之后当对项目工程中的文件有添加、删除、修改操作之后，要及时更新 VSS 数据库，以实现系统版本的控制。详细操作见后面的“使用”部分。

(4) 用户及权限的设置。

为便于根据项目中的不同成员角色对成员及其访问的资源进行分级管理，在“(1)数据库的配置”中仅对默认的用户权限进行了设置，根据实际情况应对项目中的成员分配不同的访问权限，即根据岗位角色建立项目中各个成员所对应的用户。假设目前项目中有三个成员，所以要建立三个用户：Groupleader、Programmer 和 Observer，其角色分别为：项目组长、程序员和项目监察员。项目组长应该具有全权限；程序员应当对项目文件有编辑的权限，项目监察员作为项目开发情况的监视者只有读权限即可。具体的用户权限设置如下。

① 打开 Visual SourceSafe 6.0 Admin 管理界面，单击菜单 Tools→Rights by Project... 命令，打开项目及对应用户权限管理的界面。由于在 VSS 中，用户权限具有传递性(即子

目录会继承父目录的权限), 故只需设置要控制的顶级目录即可(如果有特殊需要可以对下级目录分别进行设置), 以免去层层设置的麻烦。对应当前项目中的用户权限设置如下。

② 选择根目录下面的 VSSEExample, 可见右边 User rights 选项框中其对应的权限仅为 Read, 由于 Groupleader 是项目组长角色, 故选中其他三项 Check Out/Check In、Add/Rename/Delete 和 Destroy 复选框, 如图 A-16 所示。

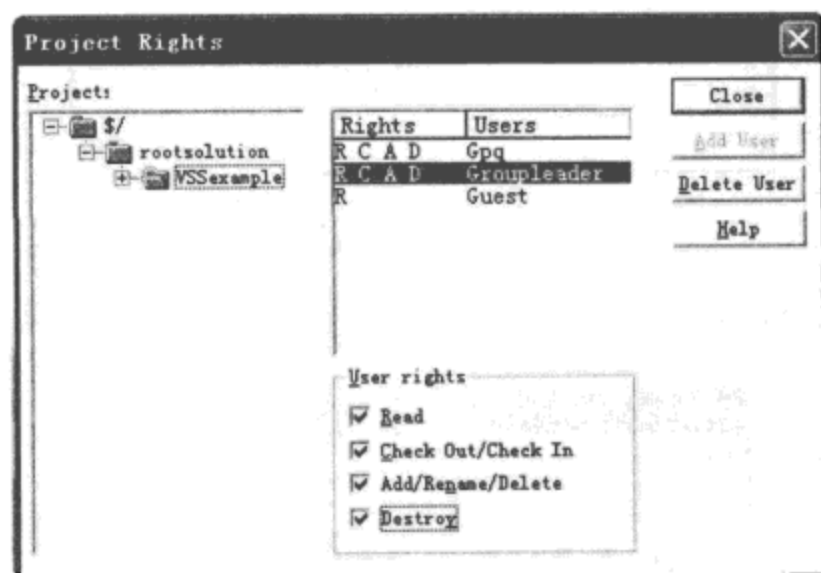


图 A-16 设置 Groupleader 的权限

同理, 可以设置用户 Programmer 的权限为“Read”、“Check Out/Check In”和“Add/Rename/Delete”; 设置“Observer”的权限为“Read”。

至此, 系统中的用户及其权限已基本设置完毕。管理员可以根据项目中的实际情况对用户及其权限设置做出相应的更改。

2) 客户端配置

以用户“User1”为例。单击“程序”→Microsoft Visual Studio 6.0→Microsoft Visual Source Safe→Microsoft Visual SourceSafe 6.0 命令, 打开登录窗口, 在用户名后输入“Groupleader”, 密码输入框中输入 Groupleader 设定的密码, 数据库框中选择刚建立的数据库 VSSTest, 单击 OK 按钮, 打开 Visual SourceSafe 客户端管理界面。在此, 便可以查看所打开的数据库中项目的当前状态, 并在权限许可的情况下对数据库做出相关的管理。

3. 使用

一旦项目进入了 VSS 的数据库, 以后所有团队成员要打开这个项目都是在这个数据库中取得工作副本, 然后在自己的本地副本上工作, 服务器上的是主控副本。团队成员只要第一次从数据库中取得数据建立本地工作副本, 以后只要跟一般的项目一样在本地打开这个项目就行了。工作副本跟主控副本通过签入签出进行交互。

1) 建立工作副本

项目进入了 VSS 的数据库, 就处于 VSS 的管理之下, 开发团队的所有成员需要从这个数据库中取得项目主控副本的本地工作副本。

在 File(文件)菜单上, 单击 Source Control(源代码管理)→Open from Source Control(从源代码管理打开)命令。系统将提示你输入相应的源代码管理数据库, 我们要找到在服务器上建立的那个 VSSTest 数据库, 单击 Browse 按钮, 在 Open SourceSafe Database 对话框中显示了你本机的 VSS 所知道的 VSS 数据库, 而 VSSTest 库并未在此列出, 我们再单击 Browse 按钮, 去找服务器上我们先前建立 VSSTest 数据库时那个共享的 VSSTest 目录, 选择在这个目录下的 srcsafe.ini 文件并打开, 给这个数据库起个名字“VSSTest”, 这样包含我们的 web 项目的 VSSTest 数据库就被引入本机的 VSS, 打开这个数据库。此数据库的登录界面如图 A-17 所示。

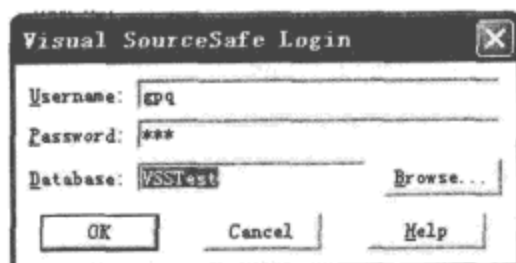


图 A-17 登录数据库

打开数据库后, 系统将提示选择这个项目的解决方案在你本机的存放路径, 如图 A-18 所示。

注意, 这里选的是解决方案的存放地, 关于解决方案在前面已经论述过, 解决方案可以放在任意地方, 当工作副本建立好以后, 正常工作就是从这个解决方案来打开工作副本进行工作的。

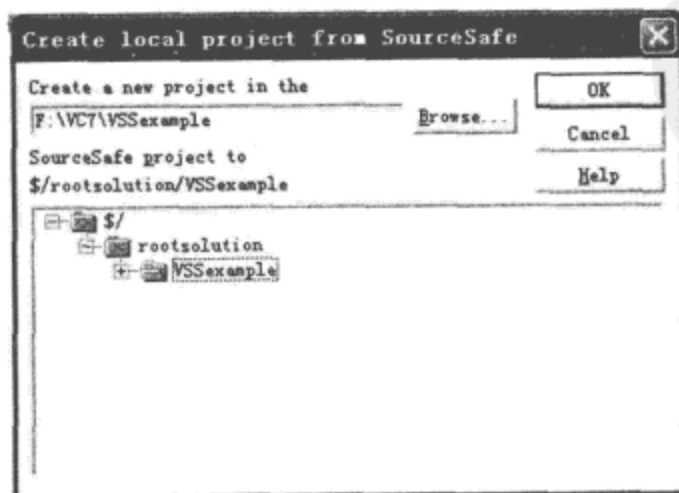


图 A-18 生成本地项目(含存放路径)

2) 访问项目文件

当在 VS.net 中的软件开发纳入 VSS 控制之下后,对项目中的每个成员都要按照一定的流程进行开发。

(1) 首先,在解决方案资源管理器中要编辑的文件或文件夹上右击,在弹出的快捷菜单中选择“获取最新版本”命令,以同步数据库中的最新版本。

(2) 在修改文件之前,应执行“签出”操作,以同步数据库上该文件的最新版本,并做标记,使其他用户只能读取而不能修改该文件(该操作基于目前服务器不支持多用户“签出”,这也是团队开发所推荐的模式)。

(3) 当对文件修改完成以后,需要对文件执行“签入”操作,将改动上传到数据库,同时释放对数据库文件修改权的锁定,对于“签入”以后的文件其他用户可以再签出进行编辑。

(4) 当要删除文件时,应先将工程文件“签出”以同步数据库上的该文件,然后执行“移除”操作,将该文件删除,最后执行“签入”操作删除数据库中的文件。



注意: ① 在文件未经“签出”时,对文件的任何修改都会导致系统显示要求将文件签出的提示。

② 文件的添加与删除涉及工程项目文件的修改,因此在添加或删除文件时需先将项目文件“签出”。

③ 虽然可以通过修改设置使多个用户同时签出同一个文件,但这可能对开发带来不良后果。因此我们建议,同一时刻,每个文件只允许一个用户做签出修改。

3) 当前工程的状态查看

在开发过程中,可以随时打开 Visual SourceSafe 对话框,以查看当前工程的状态,并做出相应的处理。单击“程序”→Microsoft Visual Studio 6.0→Microsoft Visual SourceSafe→Microsoft Visual SourceSafe 6.0 命令,以对应的用户身份打开数据库“VSSTest”即可显示。

4) 将文件恢复到以前的版本

要将文件恢复到以前的版本,可以执行以下的操作。

(1) 在 VSS 浏览器中选择要恢复的文件。

(2) 选择 Tools→Show History 命令,显示 History Options 对话框,输入相应的选取参数。

(3) 单击 OK 按钮显示 History of File 对话框。

(4) 选择文件的一个以前的版本,单击 Rollback 按钮,即可完成操作。



注意: 执行恢复操作时,会丢失掉恢复版本以后的所有对该文件的改动。例如,如果将文件恢复到版本 5,则所有从版本 5 以后的改动都会丢失。

[G e n e r a l I n f o r m a t i o n]

书名 = C + + 游戏编程

作者 = 邹吉滔编著

页码 = 4 7 2

I S B N = 4 7 2

S S 号 = 1 2 7 5 9 2 4 3

d x N u m b e r = 0 0 0 0 0 8 0 5 3 9 0 5

出版时间 = 2 0 1 1 . 0 1

出版社 = 该引擎未能查询到

定价 : 5 0 . 0 0

试读地址 = <http://book.szdnnet.org.cn/bookDetail.jsp?dxNumber=000008053905&d=E53F521F0C4B5F7577DF810451331927&fenlei=1817040302&sw=%B1%E0%B3%CC>

全文地址 = <http://nsst.5read.com/image/ss2jpg.dll?did=b59&pid=4A22422ED10B2ECFD3AD54442FF212667CF7032B8669749A25A114A0AB76BAE039F3FA88F2BC7A908DBC75158234AB53CE6E64D93FF1FC5BB1BBF24E1C265B08577F5CE354E29E4D54B4E23749BE623E93C0607D1FA24E879EDFBAD2AB6EF03B73095569729CE372BC29297B3AD6EFABAB9E&jid=/>